

1. [GAME 2302-0095: Preface to GAME 2302](#)
2. Math
 1. [GAME 2302-0100: Introduction](#)
 2. [GAME 2302-0105: Getting Started](#)
 3. [Game0105r Review](#)
 4. [GAME 2302-0110: Updating the Game Math Library for Graphics](#)
 5. [Game0110r Review](#)
 6. [GAME 2302-0115: Working with Column Matrices, Points, and Vectors](#)
 7. [Game0115r Review](#)
 8. [GAME 2302-0120: Visualizing Column Matrices](#)
 9. [Game0120r Review](#)
 10. [GAME 2302-0125: Vector Addition](#)
 11. [Game0125r: Review](#)
 12. [GAME 2302-0130: Putting the Game-Math Library to Work](#)
 13. [Game0130r: Review](#)
 14. [GAME 2302-0135: Venturing into a 3D World](#)
 15. [GAME 2302-0140: Our First 3D Game Program](#)
 16. [GAME 2302-0145: Getting Started with the Vector Dot Product](#)
 17. [GAME 2302-0150: Applications of the Vector Dot Product](#)
3. Physics
 1. [GAME 2302-0300 Introduction to Physics Component](#)
 2. [GAME 2302-0310 JavaScript](#)
 3. [GAME 2302-0320 Brief Trigonometry Tutorial](#)
 4. [GAME 2302-0330 Scale Factors, Ratios, and Proportions](#)
 5. [GAME 2302-0340 Scientific Notation and Significant Figures](#)

6. [GAME 2302-0350 Units and Dimensional Analysis](#)
7. [GAME 2302-0360 Motion -- Displacement and Vectors](#)
8. [GAME 2302-0370 Motion -- Uniform and Relative Velocity](#)
9. [GAME 2302-0380 Motion -- Variable Velocity and Acceleration](#)

GAME 2302-0095: Preface to GAME 2302

This module serves as the Preface to material for the course, GAME 2302 - Mathematical Applications for Game Development, which Prof. Baldwin teaches at Austin Community College in Austin, TX.

Welcome

Welcome to the course material for **GAME 2302 Mathematical Applications for Game Development**, which I teach at [Austin Community College](http://www.austincc.edu) in Austin, TX.

Official information about the course

The college website for this course is: <http://www.austincc.edu/baldwin/>

As of December 2012, the description for this course reads:

"GAME 2302 - Mathematical Applications for Game Development

Presents applications of mathematics and science in game and simulation programming. Includes the utilization of matrix and vector operations, kinematics, and Newtonian principles in games and simulations. Also covers code optimization."

The course material

This course material consists of about 20 different modules arranged in the following major sections:

- Math
- Physics

The modules in the Math section use Java OOP to develop a game math library.

Many of the topics consist of two separate modules:

- A tutorial (click [here](#) for an example).

- A set of review questions (click [here](#) for an example).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0095: Java OOP: Preface to GAME 2302
- File: Game0095.htm
- Published: 12/31/12
- Revised: 01/19/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0100: Introduction

This module is the first in a series of modules designed for teaching GAME2302 Mathematical Applications for Game Development at Austin Community College in Austin, TX.

Table of Contents

- [Preface](#)
- [Course description](#)
- [Adherence to the course description](#)
- [Course resources](#)
- [Homework assignments](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Download source code](#)

Preface

This module is the first in a series of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

See the [Download source code](#) section to download source code files for the modules contained in this collection.

Course description

As of September 2012, the official course description reads: *"Presents applications of mathematics and science in game and simulation programming. Includes the utilization of matrix and vector operations, kinematics, and Newtonian principles in games and simulations. Also covers code optimization."*

Adherence to the course description

Online [resources](#) are provided to the student for each of the topics identified in the above description. The topics merge in the final module of the collection, which explains an animated simulation of the first-person view of a human cannonball from the point in time that the human cannonball leaves the cannon until the human cannonball lands in the safety net, or fails to land in the safety net. The trajectory and the point of impact are determined by such factors as:

- the acceleration of gravity,
- the muzzle velocity, and
- the elevation and azimuth of the aiming mechanism on the cannon.

Course resources

This course does not use a printed textbook. Instead, the primary resources for the course are:

1. An interactive tutorial by Dr. Bradley P. Kjell titled [Vector Math for 3D Computer Graphics](#). *(You can view the tutorial online or you can download a copy of an older version of the tutorial in a zip file [here](#). Extract the contents of the zip file and open the file named **index.html** in your browser to view the tutorial offline. Be aware, however, that some of the chapters in the zip file perform animation using Java applet code that may no longer be supported.)* The tutorial is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](#).
2. A sub-collection of physics modules beginning with [2302-0300 Introduction to Physics Component](#) published at [cnx.org](#).
3. This [collection](#) of modules published at [cnx.org](#).
4. Most of the programming examples in the course will be written in Java/OOP. Therefore, you may need some reference material on Java/OOP programming. The collection named [Object-Oriented Programming.\(OOP\) with Java](#) at [cnx.org](#) contains the course material for three complete Java/OOP programming courses at Austin Community College. The section named *ITSE 2321 Object-Oriented Programming (Java)* contains the material for the first course. The section named *ITSE2317 - Java Programming (Intermediate)* contains

the material for the second course. The section named *INEW 2338 - Advanced Java (Web)* contains the material for the third course.

5. Here are some additional Java/OOP references that you may find useful:

1. [Java Platform Standard Edition 7 Documentation](#)
2. [Java Platform, Standard Edition 7 API Specification](#)
3. [Reading the Javadoc](#) - how to read the API
4. [The Java Tutorials](#)
5. [Simplified Java tutorial](#)

Main item 3 in the above list (*this [collection](#)*) will serve as the major resource for classroom lectures.

Main item 1 (*[Vector Math for 3D Computer Graphics](#)*) and main item 2 (*[physics sub-collection](#)*) will serve as major homework study assignments.

Main items 4 and 5 (*including sub-items 1 through 5*) are provided for reference only.

The material in this [collection](#) (*item 3*) explains how to implement the concepts developed in the first two items in programming (*Java*) code. While this will serve as the major resource for classroom lectures, students are encouraged to bring questions about the other items to class for discussion.

Homework assignments

The first classroom session will be dedicated primarily to explaining the mechanics of the course, discussing the syllabus, etc.

Once those items have been taken care of, students will be asked to go online, access or download the Kjell tutorials, and begin studying Kjell's *CHAPTER 0 -- Points and Lines* and *CHAPTER 1 -- Vectors, Points, and Column Matrices* down through the topic titled *Variables as Elements* in *Chapter 1* , in preparation for the next classroom session.

Students should also begin studying item 2 ([physics](#)) and should study one physics module per week thereafter.

Finally, in addition to studying the Kjell material, students should read at least the next two modules in this collection and bring their questions about that material to the next classroom session.

What's next?

In the next module, we will begin the examination of sample programs and a game-programming math library intended to provide aspiring game programmers with the mathematical skills needed for game programming.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0100: Introduction
- File: Game0100.htm
- Published: 10/08/12
- Revised: 02/09/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Download source code

Click [here](#) to download a zip file containing the source code for all of the sample programs and many of the exercises in this collection.

Extract the contents of the zip file into an empty folder. Each program should end up in a separate folder. Double-click the file named ***CompileAndRun...bat*** in each folder to compile and run the program contained in that folder.

-end-

GAME 2302-0105: Getting Started

Examine two sample programs and a sample game-programming math library intended to provide aspiring game programmers with the mathematical skills required for game programming.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [Points, lines, and vectors](#)
 - [Writing, compiling, and running Java programs](#)
 - [The program named PointLine01](#)
 - [But wait, there's something wrong here](#)
 - [The program named PointLine02 and the library named GM2D01](#)
 - [The GM2D01.ColMatrix class](#)
 - [The GM2D01.Point class](#)
 - [The GM2D01.Vector class](#)
 - [The GM2D01.Line class](#)
- [The GM2D01 library is purely mathematical](#)
- [Documentation for the GM2D01 library](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)

- [Exercise 1](#)
- [Exercise 2](#)
- [Exercise 3](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

General

Good math skills are required

In order to be a successful game programmer you must be skilled in technologies other than simply programming. Those technologies include mathematics. My purpose in writing this module is to help you to gain mathematical strengths in addition to your other strengths.

This collection is designed to teach you some of the mathematical skills that you will need (*in addition to good programming skills*) to become a successful game programmer. In addition to helping you with your math skills, I will also teach you how to incorporate those skills into object-oriented programming using Java. If you are familiar with other object-oriented programming languages such as C#, you should have no difficulty porting this material from Java to those other programming languages.

Lots of graphics

Since most computer games make heavy use of either 2D or 3D graphics, you will need skills in the mathematical areas that are required for success in 2D and 3D graphics programming. As a minimum, this includes but is not limited to skills in:

1. *Geometry*
2. *Trigonometry*

3. **Vectors**
4. **Matrices**
5. **2D and 3D transforms**
6. Transformations between coordinate systems
7. Projections

Game programming requires mathematical skills beyond those required for graphics. This collection will concentrate on items 3, 4, and 5 in the above list. *(I will assume that you either already have, or can gain the required skills in geometry and trigonometry on your own. There are many tutorials available on the web to help you in that quest including the **Brief Trigonometry Tutorial** at <http://cnx.org/content/m37435/latest/>.)*

Insofar as vectors and matrices are concerned, I will frequently refer you to an excellent interactive tutorial titled [Vector Math for 3D Computer Graphics](#) by Dr. Bradley P. Kjell for the required technical background. I will then teach you how to incorporate the knowledge that you gain from Kjell's tutorial into Java code with a heavy emphasis on object-oriented programming.

In the process, I will develop and explain a game-programming math library that you can use to experiment with and to confirm what you learn about vectors and matrices from the Kjell tutorial. The library will start out small and grow as we progress through more and more material in subsequent modules.

I will also provide exercises for you to complete on your own at the end of most of the modules. Those exercises will concentrate on the material that you have learned in that module and previous modules.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Graphics illustration of four points and two lines.
- [Figure 2](#). First of three views in the construction of a 3D human figure.
- [Figure 3](#). Second of three views in the construction of a 3D human figure.
- [Figure 4](#). Third of three views in the construction of a 3D human figure.
- [Figure 5](#). Screen output from the program named PointLine02.
- [Figure 6](#). Sample output from updated programming-math library.
- [Figure 7](#). Graphic output from Exercise 1.
- [Figure 8](#). Text output from Exercise 2.
- [Figure 9](#). Text output from Exercise 3.

Listings

- [Listing 1](#). The controlling class named PointLine01.
- [Listing 2](#). Beginning of the class named GUI.
- [Listing 3](#). Beginning of the inner class named MyCanvas.
- [Listing 4](#). Define two points as two locations in space.
- [Listing 5](#). Construct a line segment.
- [Listing 6](#). Construct an object that represents a vertical line segment.
- [Listing 7](#). Draw the two lines on the screen.
- [Listing 8](#). Beginning of the class named PointLine02.
- [Listing 9](#). Beginning of the class named GM2D01.
- [Listing 10](#). Exercising the Point class.
- [Listing 11](#). The static top-level class named Point.
- [Listing 12](#). The static top-level class named Vector.
- [Listing 13](#). The static top-level class named Line.
- [Listing 14](#). Source code for the program named PointLine01.
- [Listing 15](#). Source code for the program named PointLine02.
- [Listing 16](#). Source code for the game-programming math library named GM2D01.

Preview

In this module, I will introduce you to an excellent interactive tutorial titled [Vector Math for 3D Computer Graphics](#) written by Dr. Bradley P. Kjell, (which you can also download in zip-file format from a link in the [first module](#) in this collection) . Then I will present and explain two sample programs and a sample game-programming math library intended to implement concepts from Dr. Kjell's tutorial in Java code.

Discussion and sample code

Points, lines, and vectors

Your homework assignment from the previous module was to study *CHAPTER 0 -- Points and Lines* and also to study *CHAPTER 1 -- Vectors, Points, and Column Matrices* down through the topic titled *Variables as Elements* in Kjell's tutorial.

You may have previously believed that you already knew all there was to know about points and lines. However, I suspect that you found explanations of some subtle issues that you never thought about before when studying those tutorials. In addition, Kjell begins the discussion of vectors and establishes the relationship between a vector and a column matrix in this material.

Represent a point with a column matrix

Hopefully you understand what Kjell means by a column matrix and you understand that a column matrix can be used to represent a point in a given coordinate frame. The same point will have different representations in different coordinate frames. (*You must know which coordinate frame is being used to represent a point with a column matrix.*)

Writing, compiling, and running Java programs

One of the reasons that I chose Java as the main programming language for this course is that while Java is a very powerful object-oriented

programming language, the mechanics of writing, compiling, and running Java programs are very simple.

Confirm your Java installation

First you need to confirm that the Java development kit (*jdk*) version 1.7 or later is installed on the computer. *(The jdk is already installed in the CIT computer labs at the NRG campus of ACC, and perhaps in the labs on other ACC campuses as well.)* If you are working at home, see Oracle's [JDK 7 and JRE 7 Installation Guide](#).

Creating your source code

Next, you need to use any text editor to create your Java source code files as text files with an extension of .java. *(I prefer the free [JCreator](#) editor because it produces color-coded text and includes some other simple IDE features as well. JCreator is normally installed in the CIT computer labs at the NRG campus of ACC.)*

Compiling your source code

The name of each source code file should match the name of the Java class defined in the file.

Assume that your source code file is named **MyProg.java** . You can compile the program by opening a command prompt window in the folder containing the source code file and executing the following command at the prompt:

Note: Command to compile the source code:

```
javac MyProg.java
```

Running your Java program

Once the program is compiled, you can execute it by opening a command prompt window in the folder containing the compiled source code files (*files with an extension of .class*) and executing the following command at the prompt:

Note: Command to execute the program:

```
java MyProg
```

Using a batch file

If you are running under Windows, the easiest approach is to create and use a batch file to compile and run your program. (*A batch file is simply a text file with an extension of .bat instead of .txt.*)

Create a text file named **CompileAndRun.bat** containing the text shown in the note-box below.

Note: Contents of batch file are shown below:

```
del *.class  
javac MyProg.java  
java MyProg  
pause
```

Place this file in the same folder as your source code files. Then double-click on the batch file to cause your program to be compiled and executed.

That's all there is to it.

The program named **PointLine01**

Before we go any further, let's take a look at a simple Java program that illustrates one of the ways that points and lines are represented in Java code. (See [Figure 1](#).)

The **Point2D.Double** class

This program illustrates one implementation of the concepts of *point* and *line segment* in Java code.

Four points (*locations in space*) are defined by passing the coordinates of the four points as the x and y parameters to the constructor for the **Point2D.Double** class. This results in four objects of the **Point2D.Double** class.

(*Point2D.Double* is a class in the standard Java library.)

The **Line2D.Double** class

Two line segments are defined by passing pairs of points as parameters to the constructor for the **Line2D.Double** class. This results in two objects of the **Line2D.Double** class.

(*Line2D.Double* is a class in the standard Java library.)

Note: Testing:

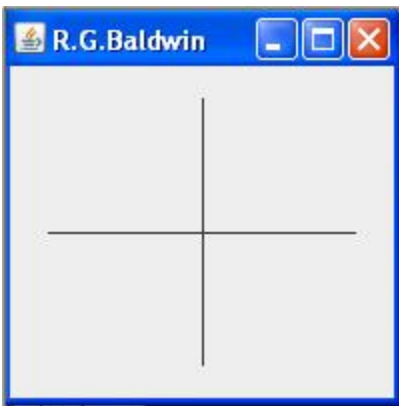
All of the programs in this module were tested using JDK 1.7 running under Windows XP.

The draw method

The **draw** method belonging to an object of the **Graphics2D** class is used to draw the two line segments on a **Canvas** object for which the origin has been translated to the center of the **Canvas** . The result is shown in [Figure 1](#) .

(*Graphics2D and Canvas are classes in the standard Java library.*)

Figure 1 Graphics illustration of four points and two lines.



The coordinate values of the points and the selection of point-pairs to specify the ends of the line segments is such that the final rendering is a pair of orthogonal lines that intersect at the origin.

(*You could think of these lines as the axes in a Cartesian coordinate system.*)

Will explain in fragments

I will present and explain this program in fragments. A complete listing of the program is provided in [Listing 14](#) near the end of the module.

(*Use the code in [Listing 14](#) and the instructions provided [above](#) to write, compile, and run the program. This will be a good way for you to confirm that Java is properly installed on your computer and that you are able to follow the instructions to produce the output shown in [Figure 1](#) .*)

The first code fragment is shown in [Listing 1](#) .

Listing 1 . The controlling class named PointLine01.

```
class PointLine01{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class PointLine01
```

[Listing 1](#) shows the definition of the controlling class, including the **main** method for the program named **PointLine01** .

The **main** method simply instantiates a new object of a class named **GUI** and saves that object's reference in the variable named **guiObj**.

(GUI is not a class in the standard Java library. It is defined below.)

The **GUI** object produces the screen image shown in [Figure 1](#).

The class named GUI

[Listing 2](#) shows the beginning of the class named **GUI** , including the constructor for the class.

Listing 2 . Beginning of the class named GUI.

Listing 2 . Beginning of the class named GUI.

```
class GUI extends JFrame{
    //Specify the horizontal and vertical size of
    a JFrame
    // object.
    int hSize = 200;
    int vSize = 200;

    GUI(){//constructor

        //Set JFrame size and title
        setSize(hSize,vSize);
        setTitle("R.G.Baldwin");

        //Create a new drawing canvas and add it to
the
        // center of the JFrame.
        MyCanvas myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setVisible(true);

    }//end constructor
```

[Listing 2](#) begins by declaring and initializing a pair of instance variables that will be used to specify the size of the **JFrame** shown in [Figure 1](#).

*(A **JFrame** object in Java might be called a window in other programming environments.)*

The constructor for the class named GUI

The constructor begins by using the two instance variables to set the size of the **JFrame** and by setting the title for the **JFrame** to *R.G.Baldwin* .

Create a new drawing canvas and add it to the center of the JFrame

Then the constructor instantiates a **Canvas** object and adds it to the **JFrame** . Without getting into the details as to why, I will simply tell you that the **Canvas** object fills the entire client area of the frame, which is the area inside of the four borders.

Define the behavior of the X-button and make the JFrame visible

Finally, the constructor defines the behavior of the X-button in the top right corner of the **JFrame** and causes the **JFrame** to become visible on the screen.

Hopefully, everything that I have discussed so far is familiar to you. If not, you may want to spend some time studying my earlier tutorials in this area. You will find links to many of those tutorials at www.DickBaldwin.com .

Beginning of the inner class named MyCanvas

[Listing 3](#) shows the beginning of an inner class definition named **MyCanvas** including the beginning of an overridden **paint** method.

*(The **Canvas** class was extended into the **MyCanvas** class to make it possible to override the **paint** method.)*

This overridden **paint** method will be executed whenever the **JFrame** containing the **Canvas** is displayed on the computer screen, whenever that portion of the screen needs to be repainted, or whenever the **repaint** method is called on the canvas.

Listing 3 . Beginning of the inner class named MyCanvas.

```
class MyCanvas extends Canvas{

    public void paint(Graphics g){
        //Downcast the Graphics object to a
Graphics2D
        // object. The Graphics2D class provides
        // capabilities that don't exist in the
Graphics
        // class.
        Graphics2D g2 = (Graphics2D)g;

        //By default, the origin is at the
upper-left corner
        // of the canvas. This statement
translates the
        // origin to the center of the canvas.
        g2.translate(

this.getWidth()/2.0,this.getHeight()/2.0);
```

An object of the Graphics class

The **paint** method always receives an incoming reference to an object of the class **Graphics** . As a practical matter, you can think of that object as representing a rectangular area on the screen on which the instructions in the **paint** method will draw lines, images, and other graphic objects.

Note: Incoming reference:

The paint method actually receives a reference to an object of the class **Graphics2D** , but to maintain backward compatibility, Sun elected to pass it in as the superclass type **Graphics** .

The Graphics2D class

The **Graphics** class was defined early in the life of the Java programming language, and it is of limited capability. Later on, Sun defined the **Graphics2D** class as a subclass of **Graphics** . Significant new capabilities, *(including the ability to deal with coordinate values as real numbers instead of integers)* , were included in the **Graphics2D** class. In order to access those new capabilities, however, it is necessary to downcast the incoming reference to type **Graphics2D** as shown by the cast operator (**Graphics2D**) in [Listing 3](#).

Translate the origin

The last statement in [Listing 3](#) translates the origin from the default position at the upper-left corner of the canvas to the center of the canvas. This is equivalent to creating a new *coordinate frame* as explained by Kjell.

The default coordinate frame has the origin at the upper-left corner of the canvas. The new coordinate frame has the origin at the center of the canvas.

Note, however, that even though the origin has been translated, the positive vertical direction is still toward the bottom of the canvas.

Define two points as two locations in space

[Listing 4](#) defines two points in space by instantiating two objects of the class named **Point2D.Double** , and saving references to those objects in variables of the superclass type **Point2D** .

Listing 4 . Define two points as two locations in space.

Listing 4 . Define two points as two locations in space.

```
Point2D pointA =  
    new Point2D.Double(-  
this.getWidth()/2.5,0.0);  
Point2D pointB =  
    new  
Point2D.Double(this.getWidth()/2.5,0.0);
```

The left and right ends of the horizontal line

The object referred to as **pointA** represents the location in 2D space at the left end of the horizontal line shown in [Figure 1](#).

As explained by Kjell, the values used to represent this location in space are relative to the current coordinate frame in which the origin is at the center of the canvas. Had the origin not been translated to the center of the canvas in [Listing 3](#), a different set of values would have been required to represent that same location in space.

Similarly, the object referred to as **pointB** represents the location in space at the right end of the horizontal line shown in [Figure 1](#).

Construct a line segment

[Listing 5](#) uses the two points described above to construct an object that represents a line segment connecting the two points by instantiating an object of the class named **Line2D.Double** .

As shown in [Figure 1](#), the values of the points relative to the current coordinate frame causes this line segment to be horizontal when the coordinate frame is displayed with the orientation shown in [Figure 1](#). On the other hand, a different rendering of the coordinate frame may have caused the line segment to be something other than horizontal.

Listing 5 . Construct a line segment.

```
Line2D.Double horizLine =  
                        new  
Line2D.Double(pointA, pointB);
```

Not the true line segment

The main point here is that the line segment created in [Listing 5](#) represents a line segment between two points in space but it is not the true line segment. According to Kjell, the true line segment has no width, and therefore is not visible to the human eye.

The line segment constructed in [Listing 5](#) is simply a visual representation of the true line segment. When rendered on the screen, that line segment could take on an infinite number of appearances depending on how the space is rendered on the screen. It just happens that in the case of [Figure 1](#), the rendering of that space on the screen causes the line segment to appear to be parallel to the bottom of a rectangular screen.

Construct an object that represents a vertical line segment

[Listing 6](#) uses a similar procedure to construct an object that represents a line segment that is perpendicular to the previous line segment, and which intersects the previous line segment at the origin of the current coordinate frame.

Listing 6 . Construct an object that represents a vertical line segment.

Listing 6 . Construct an object that represents a vertical line segment.

```
        Point2D pointC =
            new Point2D.Double(0.0, -
this.getHeight()/2.5);
        Point2D pointD =
            new
Point2D.Double(0.0, this.getHeight()/2.5);

        Line2D.Double vertLine =
            new
Line2D.Double(pointC, pointD);
```

Draw the two lines on the screen

Finally, [Listing 7](#) calls the **draw** method of the **Graphics2D** class twice in succession to draw the two lines on the screen as shown in [Figure 1](#).

Listing 7 . Draw the two lines on the screen.

```
        g2.draw(horizLine);
        g2.draw(vertLine);

    } //end overridden paint()
} //end inner class MyCanvas
} //end class GUI
```

[Listing 7](#) also signals the end of the overridden **paint** method, the end of the inner class named **MyCanvas** , and the end of the top-level class named **GUI** .

But wait, there's something wrong here

Kjell describes his points and lines in a very general way that is suitable for use in mathematical operations later on. However, in the above program, I fell into the trap of defining my points and lines using Java classes that are intended primarily for rendering graphics on the screen. That approach is probably not conducive to mathematical operations. We need to step back and take another look at what we are doing here.

Points, points, and more points

There will be many occasions when you, as a game programmer, will need to define the coordinate values for a point (*or a set of points*) that you have no intention of displaying on the screen. Instead, you will use those points for various mathematical operations to produce something else that may or may not be displayed on the screen.

The Alice ice skater

For example, [Figure 2](#), [Figure 3](#), and [Figure 4](#) show three views in the construction of the Ice skater object in the [Alice programming language](#).

Figure 2 First of three views in the construction of a 3D human figure.



Figure 3 Second of three views in the construction of a 3D human figure.



Figure 4 Third of three views in the construction of a 3D human figure.



Points as the vertices of triangles

The image in [Figure 2](#) shows a graphical representation of a set of points. These points were used to define the vertices of the set of polygons shown in [Figure 3](#). The polygons, in turn, were used in the construction of the ice skater object shown in [Figure 4](#).

As you can see, neither the points in [Figure 2](#), nor the lines that comprise the sides of the polygons in [Figure 3](#) appear in the final rendering of the object in [Figure 4](#).

However, both the points and the polygons were required to support the mathematical operations that ultimately resulted in the ice skater object.

(Only the image in [Figure 4](#) is typically displayed on the computer screen. I had to do some extra work to cause the points and the lines to be displayed.)

Another look at points and lines

We'll take another look at points and lines, and will introduce column matrices and vectors in the next sample program.

What is a vector?

According to Kjell, "A vector is a geometrical object that has two properties: length and direction." He also tells us, "A vector does not have a position."

In addition, Kjell tells us that we can represent a vector with two real numbers in a 2D system and with three real numbers in a 3D system.

Use a column matrix to represent a vector

A column matrix provides a good way to store two real numbers in a computer program. Therefore, in addition to representing a point, a column matrix can also be used to represent a vector.

However, ***the column matrix is not the vector***. The contents of the column matrix simply represent certain attributes of the vector in a particular reference frame. Different column matrices can be used to represent the

same vector in different reference frames, in which case, the contents of the matrices will be different.

An absolute location in space

The fact that a column matrix can be used to represent both points and vectors can be confusing. However, as you will see later, this is convenient from a programming viewpoint.

The two (*or three*) real number values contained in the matrix to represent a point specify an absolute location in space relative to the current coordinate frame.

A vector specifies a displacement

A vector does not have a position. Rather, it has only two properties: length and direction. Kjell tells us that the two (*or three*) real number values contained in the matrix to represent a vector (*in 2D or 3D*) specify a *displacement* of a specific distance from an arbitrary point in a specific direction.

In 2D, the two values contained in the matrix represent the displacements along a pair of orthogonal axes (*call them x and y for simplicity*) .

As you will see in a future module, in the case of 2D, the length of the vector is the length of the hypotenuse of a right triangle formed by the x and y displacement values.

The direction of the vector can be determined from the angle formed by the x -displacement and the line segment that represents the hypotenuse of the right triangle. Similar considerations apply in 3D as well but they are somewhat more complicated.

The bottom line is that while a point is an *absolute location* , a vector is a *displacement* .

Do we need to draw vectors?

It is very common to draw vectors in various engineering disciplines, such as when drawing free-body diagrams in theoretical mechanics. My guess is that it is unusual to draw vectors in the final version of computer games, but I may be wrong.

Normally what you will need to draw in a computer game is the result of one or more vectors acting on an object, such as the velocity and acceleration vectors that apply to a speeding vehicle going around a curve. In that case, you might draw the results obtained from using the vector for mathematical computations (*perhaps the vehicle turns over*) but you probably wouldn't draw the vectors themselves.

The program named PointLine02 and the library named GM2D01

The purpose of this program is to introduce you to a game-math library named **GM2D01** .

(The class name GM2D01 is an abbreviation for GameMath2D01. Later in this collection, I will develop and present a combination 2D/3D game-math library named GM03. I will develop and present several intermediate 2D and 3D libraries along the way.)

This program instantiates objects from the following **static** top-level classes belonging to the class named **GM2D01** :

- GM2D01.ColMatrix
- GM2D01.Line
- GM2D01.Point
- GM2D01.Vector

(See the [documentation](#) for the library named **GM2D01** .)

Then the program displays the contents of those objects on the standard output device in two different ways.

A complete listing of the program named **PointLine02** is provided in [Listing 15](#) near the end of the module and a complete listing of the game-

math library named **GM2D01** is provided in [Listing 16](#).

Screen output from the program named PointLine02

[Figure 5](#) shows the screen output produced by running this program. I will refer back to the material in [Figure 5](#) in subsequent paragraphs.

Figure 5 . Screen output from the program named PointLine02.

Figure 5 . Screen output from the program named PointLine02.

Instantiate and display the contents
of a new ColMatrix object

2.5,6.8

2.5

6.8

Bad index

Instantiate and display the contents
of a new Point object

3.4,9.7

3.4

9.7

Bad index

Instantiate and display the contents
of a new Vector object

-1.9,7.5

-1.9

7.5

Bad index

Instantiate and display the contents
of a new Line object

Tail = 1.1,2.2

Head = 3.3,4.4

1.1,2.2

3.3,4.4

Press any key to continue...

The game-math library named GM2D01

As mentioned earlier, the name **GM2D01** is an abbreviation for GameMath2D01. *(I elected to use the abbreviated name to keep the code*

from being so long.) This is a game-math class, which will be expanded over time as I publish sample programs and additional modules in this collection.

For educational purposes only

The game-math class is provided solely for educational purposes. Although some things were done to optimize the code and make it more efficient, the class was mainly designed and implemented for maximum clarity. Hopefully the use of this library will help you to better understand the programming details of various mathematical operations commonly used in graphics, game, and simulation programming.

Each time the library is expanded or modified, it will be given a new name by incrementing the two digits at the end of the class name to make one version distinguishable from the next. No attempt will be made to maintain backward compatibility from one version of the library to the next.

Static top-level methods

The **GM2D01** class contains several static top-level classes (See [Java 2D Graphics, Nested Top-Level Classes and Interfaces](#) if you are unfamiliar with static top-level classes.) . This organizational approach was chosen primarily for the purpose of gathering the individual classes together under a common naming umbrella while avoiding name conflicts within a single package.

For example, as time passes and this library is expanded, my default package may contain class files with the following names, each representing a compiled version of the **Point** class in a different version of the overall library.

- GM2D01\$Point.class
- GM2D02\$Point.class
- GM2D03\$Point.class

Real numbers represented as type double

All real-number values used in the library are maintained as type **double** because **double** is the default representation for literal real numbers in Java.

Will explain in fragments

I will explain the code in [Listing 15](#) and [Listing 16](#) in fragments, switching back and forth between the code from the program named **PointLine02** and the library class named **GM2D01** to show how they work together.

Beginning of the class named PointLine02

[Listing 8](#) shows the beginning of the class named **PointLine02** including the beginning of the **main** method.

Listing 8 . Beginning of the class named PointLine02.

Listing 8 . Beginning of the class named PointLine02.

```
class PointLine02{
    public static void main(String[] args){

        System.out.println(
            "Instantiate and display the
contents\n"
            + "of a new ColMatrix
object");

        GM2D01.ColMatrix colMatrix =
            new
GM2D01.ColMatrix(2.5,6.8);

        System.out.println(colMatrix);
        try{

System.out.println(colMatrix.getData(0));

System.out.println(colMatrix.getData(1));

            //This statement will throw an exception
on purpose

System.out.println(colMatrix.getData(2));
        }catch(Exception e){
            System.out.println("Bad index");
        }//end catch
    }
}
```

The GM2D01.ColMatrix class

You learned about a column matrix in the Kjell tutorial. The **GM2D01** class contains a class named **ColMatrix** . *(You will see the code for that class*

definition later.) An object of the **ColMatrix** class is intended to represent a column matrix as described by Kjell.

The code in [Listing 8](#) instantiates and displays the contents of a new object of the **ColMatrix** class. *(Note the syntax required to instantiate an object of a static top-level class belonging to another class as shown in [Listing 8](#).)*

After instantiating the object, the remaining statements in [Listing 8](#) display the numeric contents of the **ColMatrix** object using two different approaches.

The overridden **toString** method

The first approach causes the overridden **toString** method belonging to the **ColMatrix** class to be executed.

*(The overridden **toString** method is executed automatically by the call to the **System.out.println** method, passing the object's reference as a parameter to the **println** method.)*

The overridden **toString** method returns a string that is displayed on the standard output device. That string contains the values of the two real numbers stored in the column matrix.

The **getData** method

The second approach used to display the data in [Listing 8](#) calls the **getData** method on the **ColMatrix** object twice in succession to get the two numeric values stored in the object and to display those two values on the standard output device.

As you will see shortly, the **getData** method requires an incoming index value of either 0 or 1 to identify the numeric value that is to be returned.

[Listing 8](#) purposely calls the **getData** method with an index value of 2 to demonstrate that this will cause the method to throw an **IndexOutOfBoundsException** .

(The text output produced by the code in [Listing 8](#) is shown near the top of [Figure 5](#).)

Beginning of the class named GM2D01

[Listing 9](#) shows the beginning of the library class named **GM2D01** , including the entire static top-level class named **ColMatrix** .

Listing 9 . Beginning of the class named GM2D01.

Listing 9 . Beginning of the class named GM2D01.

```
public class GM2D01{
    public static class ColMatrix{
        double[] data = new double[2];

        ColMatrix(double data0,double data1)
    { //constructor
        data[0] = data0;
        data[1] = data1;
    } //end constructor

    public String toString(){
        return data[0] + "," + data[1];
    } //end overridden toString method

    public double getData(int index){
        if((index < 0) || (index > 1))
            throw new
IndexOutOfBoundsException();
        return data[index];
    } //end getData

    } //end class ColMatrix
```

The GM2D01.ColMatrix class

As mentioned earlier, an object of the **ColMatrix** class represents a 2D column matrix as described by Kjell. Furthermore, an object of this class is the fundamental building block for several of the other classes in the library.

*(The static **ColMatrix** class will be expanded in future modules to provide various matrix arithmetic capabilities.)*

Constructor for the ColMatrix class

The constructor for the class requires two incoming parameters of type **double** .

(For this example, the code in [Listing 8](#) passes the values 2.5 and 6.8 to the constructor for the class.)

The two incoming parameter values are stored in the first two elements of a two-element array of type **double** where they can be easily accessed later for whatever purpose they may be needed.

Overridden toString method

[Listing 9](#) also overrides the **toString** method to construct and return a reference to a **String** object containing the two values stored in the array.

When the **println** method is called for the second time, *(near the middle of [Listing 8](#))*, the overridden **toString** method is called automatically and the output shown in the third line of text in [Figure 5](#) is produced.

The getData method

Finally, [Listing 9](#) defines a method named **getData** . The purpose of this method is to retrieve and to return the individual values stored in the array.

The method requires an incoming parameter value of 0 or 1. This value is used as an index to identify the specific data value that is to be returned. If the method receives any other value, it throws an **IndexOutOfBoundsException** .

As mentioned earlier, the code in [Listing 8](#) calls this method three times in succession. The first two calls get and display the two data values shown at the top of [Figure 5](#). The third call causes the method to throw an exception producing the first "Bad index" message shown in [Figure 5](#).

The Point class

The **GM2D01** class contains a static top-level class named **Point** . Recall that Kjell tells us that a *point* is simply a location in space. A point can be represented by a pair of coordinate values in a specific coordinate frame. A convenient way to handle the pair of coordinate values in a program is to store them in a column matrix. An object of the **GM2D01.Point** class is intended to represent a point in 2D space.

Instantiating a Point object

As you will see shortly, the constructor for the **Point** class requires a single incoming parameter, which is a reference to an object of the class **ColMatrix** .

[Listing 10](#), (which is a fragment from the *PointLine02* program) , instantiates a new **ColMatrix** object and populates it with the values 3.4 and 9.7. Then it instantiates a new **Point** object, passing the aforementioned **ColMatrix** object's reference as a parameter to the **Point** constructor.

Listing 10 . Exercising the Point class.

Listing 10 . Exercising the Point class.

```
//A fragment from the PointLine02 program

    System.out.println(/*blank line*/);
    System.out.println(
        "Instantiate and display the
contents\n"
        + "of a new Point object");

    colMatrix = new GM2D01.ColMatrix(3.4,9.7);
    GM2D01.Point point = new
GM2D01.Point(colMatrix);

    System.out.println(point);
    try{
        System.out.println(point.getData(0));
        System.out.println(point.getData(1));
        //This statement will throw an exception
on purpose
        System.out.println(point.getData(-1));
    }catch(Exception e){
        System.out.println("Bad index");
    }//end catch
```

Coding for clarity

Note: Coding for clarity:

Ordinarily I would compress those two statements into a single statement by instantiating an anonymous object of the ColMatrix class in the call to the constructor for the Point class, and I recommend that you do the same if you know how. However, I elected to separate the code into two

statements in this case to provide clarity and make it somewhat easier for you to understand.

Display the values

Following that, the code in [Listing 10](#) displays the coordinate values that represent the point in the same two ways described earlier for the **ColMatrix** object. The screen output is shown in [Figure 5](#).

The GM2D01.Point class

The complete definition of the static top-level class named **Point** is shown in [Listing 11](#).

Listing 11 . The static top-level class named Point.

Listing 11 . The static top-level class named Point.

```
//A fragment from the GM2D01 class

public static class Point{
    GM2D01.ColMatrix point;

    Point(GM2D01.ColMatrix point){
        this.point = point;
    }//end constructor

    public String toString(){
        return point.getData(0) + "," +
point.getData(1);
    }//end toString

    public double getData(int index){
        if((index < 0) || (index > 1))
            throw new
IndexOutOfBoundsException();
        return point.getData(index);
    }//end getData

} //end class Point
```

Less detail in the discussions

By now, you may be getting a little bored with the detail in which I have been discussing and explaining the code so I will be more brief in my explanations from here on.

The code in [Listing 11](#) is straightforward and shouldn't require a lot of explanation. Perhaps the most significant thing to note about [Listing 11](#) is that the coordinate values that represent the point are actually stored

internally in an object of the type **ColMatrix** . That approach will prove to be convenient for certain mathematical operations that will be explained in future modules.

The GM2D01.Vector class

The static top-level class named **Vector** is shown in [Listing 12](#).

*(Note that this is a different class from the class named **java.util.Vector** in the standard Java library.)*

You will find the code that exercises this class and produces the output shown in [Figure 5](#) in the complete listing of the program named **PointLine02** in [Listing 15](#). That code is straightforward and shouldn't require an explanation.

Listing 12 . The static top-level class named Vector.

Listing 12 . The static top-level class named Vector.

```
//A fragment from the GM2D01 class

public static class Vector{
    GM2D01.ColMatrix vector;

    Vector(GM2D01.ColMatrix vector){
        this.vector = vector;
    }//end constructor

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1);
    }//end toString

    public double getData(int index){
        if((index < 0) || (index > 1))
            throw new
IndexOutOfBoundsException();
        return vector.getData(index);
    }//end getData

} //end class Vector
```

Storing a vector in a column matrix

Recall that Kjell tells us that both points and vectors can be conveniently stored in a column matrix. As a result, the code in [Listing 12](#), (*at this stage in the development of the library*) , is essentially the same as the code for the **Point** class in [Listing 11](#). The only differences are a few differences in names.

You may be wondering why I didn't simply define a single class that can serve both purposes. I probably could have done that. Recall however that this library is being designed for clarity. I believe that such clarity is best served by having consistent names for the kinds of items represented by objects of the classes. Also, it is likely that the definitions of the two classes will be different later when I expand the library to provide additional capabilities.

The GM2D01.Line class

Kjell tells us that a line segment is the straight path between two points, and that it has no thickness. The class named **GM2D01** contains a class named **Line** that is intended to represent a mathematical line segment as described by Kjell.

[Listing 13](#) is a complete listing of the class named **Line**. As before, you will find the code that exercises this class and produces the output shown in [Figure 5](#) in the complete listing for the program named **PointLine02** in [Listing 15](#). That code is straightforward and shouldn't require an explanation.

Listing 13 . The static top-level class named Line.

Listing 13 . The static top-level class named Line.

```
//A fragment from the GM2D01 class

//A line is defined by two points. One is
called the
// tail and the other is called the head.
public static class Line{
    GM2D01.Point[] line = new GM2D01.Point[2];

    Line(GM2D01.Point tail,GM2D01.Point head){
        this.line[0] = tail;
        this.line[1] = head;
    }//end constructor

    public String toString(){
        return "Tail = " + line[0].getData(0) +
", "
        + line[0].getData(1) + "\nHead =
"
        + line[1].getData(0) + ", "
        + line[1].getData(1);
    }//end toString

    public GM2D01.Point getTail(){
        return line[0];
    }//end getTail

    public GM2D01.Point getHead(){
        return line[1];
    }//end getTail

} //end class Line
```

Represent a line segment by two points

Since a line segment is the straight path between two points, a line segment in this library is represented by an object that encapsulates two **Point** objects. One of those points is referred to as the *tail* and the other is referred to as the *head* , simply as a means of distinguishing between the two ends of the line segment.

The constructor for the **Line** class requires two points as incoming parameters and stores them in a two-element array of type **GM2D01.Point** . Beyond that, the code in [Listing 13](#) is straightforward and shouldn't require further explanation.

The GM2D01 library is purely mathematical

The library named **GM2D01** is purely mathematical. By that, I mean that the library doesn't provide any mechanism for rendering objects of the **ColMatrix** , **Line** , **Point** , or **Vector** classes in a visual graphics context. That capability will be added to the next version of the library in the next module.

Documentation for the GM2D01 library

Click [here](#) to download a zip file containing standard javadoc documentation for the library named **GM2D01** . Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text (see [Listing 16](#) and the explanations given above) , the documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

Homework assignment

Your homework assignment for this module was to study Kjell's *CHAPTER 0 -- Points and Lines* plus *CHAPTER 1 -- Vectors, Points, and Column Matrices* down through the topic titled *Variables as Elements* .

The homework assignment for the next module is to make certain that you have carefully studied that material, and to mentally reflect on how it correlates with what you have learned in this module.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 14](#), [Listing 15](#), and [Listing 16](#) into your text editor. Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make sure you understand why your changes produce the results that they do.

Summary

In this and the previous module, I introduced you to an excellent interactive tutorial titled [Vector Math for 3D Computer Graphics](#) written by Dr. Bradley P. Kjell. Then I presented and explained two sample programs and a sample game-programming math library intended to represent concepts from Dr. Kjell's tutorial in Java code.

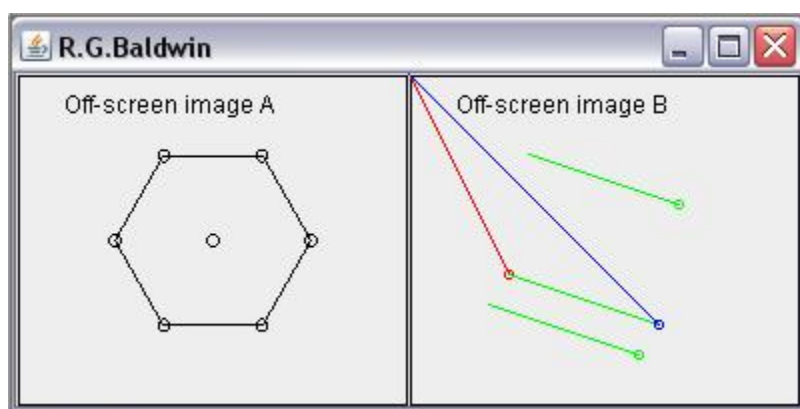
What's next?

While you probably won't have a frequent need to present points, lines, and vectors in graphical form in computer games that you write, it is often very

useful to provide graphical representations of these items during the testing and debugging of the program. I will update the programming-math library to make it easy to provide graphical representations of points, lines, and vectors in the next module in this collection.

An example of such graphical output is shown in [Figure 6](#). The image on the left consists of graphical objects that represent points and lines. The image on the right consists of graphical objects that represent vectors. (*The head of each vector is represented by a small circle.*)

Figure 6 Sample output from updated programming-math library.



Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0105: Getting Started
- File: Game0105.htm
- Published: 10/13/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed above are provided in [Listing 14](#), [Listing 15](#), and [Listing 16](#) below.

Listing 14 . Source code for the program named PointLine01.

```
/*PointLine01.java  
Copyright 2008, R.G.Baldwin
```

This program illustrates the implementation of the concept of a point and a line segment in Java.

Four points (locations in space) are defined by using the coordinates of the four points as the x and y

parameters
to the constructor for the Point2D.Double class.
This
results in four objects of the Point2D.Double
class.

Two line segments are defined by using pairs of
points
as parameters to the Line2D.Double class. This
results
in two objects of the Line2D.Double class.

The draw method belonging to an object of the
Graphics2D
class is used to draw the two line segments on a
Canvas
object for which the origin has been translated to
the
center of the Canvas.

The coordinate values of the points and the
selection of
point-pairs to specify the ends of the line
segments is
such that the final rendering is a pair of
orthogonal
lines that intersect at the origin.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/  
import java.awt.geom.*;  
import java.awt.*;  
import javax.swing.*;  
  
class PointLine01{  
    public static void main(String[] args){
```

```

        GUI guiObj = new GUI();
    }//end main
} //end controlling class PointLine01
//=====
=====//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 200;
    int vSize = 200;

    GUI(){//constructor

        //Set JFrame size and title
        setSize(hSize,vSize);
        setTitle("R.G.Baldwin");

        //Create a new drawing canvas and add it to
the
        // center of the JFrame.
        MyCanvas myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setVisible(true);

    } //end constructor
    //-----
    -----//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{

```

```

        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas in
its
        // contentPane are displayed on the screen.
        public void paint(Graphics g){
            //Downcast the Graphics object to a
Graphics2D
            // object. The Graphics2D class provides
            // capabilities that don't exist in the
Graphics
            // class.
            Graphics2D g2 = (Graphics2D)g;

            //By default, the origin is at the upper-
left corner
            // of the canvas. This statement translates
the
            // origin to the center of the canvas.
            g2.translate(

this.getWidth()/2.0,this.getHeight()/2.0);

            //Define two points.
            Point2D pointA =
                new Point2D.Double(-
this.getWidth()/2.5,0.0);
            Point2D pointB =
                new
Point2D.Double(this.getWidth()/2.5,0.0);
            //Use the points to construct an object that
            // represents a line segment that connects
the two
            // points. The values of the points causes
this
            // line segment to be horizontal.
            Line2D.Double horizLine =

```

```

                                new
Line2D.Double(pointA,pointB);

    //Use the same procedure to construct an
    object that
    // represents a vertical line segment.
    Point2D pointC =
        new Point2D.Double(0.0, -
this.getHeight()/2.5);
    Point2D pointD =
        new
Point2D.Double(0.0,this.getHeight()/2.5);
    Line2D.Double vertLine =
                                new
Line2D.Double(pointC,pointD);

    //Draw the horizontal and vertical line
    segments on
    // the canvas.
    g2.draw(horizLine);
    g2.draw(vertLine);

    }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI

```

Listing 15 . Source code for the program named PointLine02.

```

/*PointLine02.java
Copyright 2008, R.G.Baldwin

```

The purpose of this program is to introduce the use of a game-math class library named GM2D01. The class name

GM2D01 is an abbreviation for GameMath2D01.

The program instantiates objects from the following static top-level classes belonging to the class named GM2D01 and then displays the contents of those objects in two different ways on the standard output device.

ColMatrix
Point
Vector
Line

Tested using JDK 1.6 under WinXP.

*****/

```
class PointLine02{
    public static void main(String[] args){

        System.out.println(
            "Instantiate and display the
contents\n"
            + "of a new ColMatrix object");
        GM2D01.ColMatrix colMatrix =
            new
GM2D01.ColMatrix(2.5,6.8);
        System.out.println(colMatrix);
        try{
            System.out.println(colMatrix.getData(0));
            System.out.println(colMatrix.getData(1));
            //This statement will throw an exception on
purpose
            System.out.println(colMatrix.getData(2));
        }catch(Exception e){
            System.out.println("Bad index");
        }//end catch
    }
}
```

```

-

System.out.println(/*blank line*/);
System.out.println(
    "Instantiate and display the
contents\n"
    + "of a new Point object");
colMatrix = new GM2D01.ColMatrix(3.4,9.7);
GM2D01.Point point = new
GM2D01.Point(colMatrix);
System.out.println(point);
try{
    System.out.println(point.getData(0));
    System.out.println(point.getData(1));
    //This statement will throw an exception on
purpose
    System.out.println(point.getData(-1));
}catch(Exception e){
    System.out.println("Bad index");
} //end catch

System.out.println(/*blank line*/);
System.out.println(
    "Instantiate and display the
contents\n"
    + "of a new Vector object");
colMatrix = new GM2D01.ColMatrix(-1.9,7.5);

GM2D01.Vector vector = new
GM2D01.Vector(colMatrix);
System.out.println(vector);
try{
    System.out.println(vector.getData(0));
    System.out.println(vector.getData(1));
    //This statement will throw an exception on
purpose
    System.out.println(vector.getData(2));
}catch(Exception e){
    System.out.println("Bad index");
}

```

```

    }//end catch

    System.out.println(/*blank line*/);
    System.out.println(
        "Instantiate and display the
contents\n"
        + "of a new Line object");
    GM2D01.ColMatrix colMatrixTail =
        new
GM2D01.ColMatrix(1.1,2.2);
    GM2D01.ColMatrix colMatrixHead =
        new
GM2D01.ColMatrix(3.3,4.4);

    GM2D01.Point pointTail =
        new
GM2D01.Point(colMatrixTail);
    GM2D01.Point pointHead =
        new
GM2D01.Point(colMatrixHead);

    GM2D01.Line line =
        new
GM2D01.Line(pointTail,pointHead);
    System.out.println(line);

    pointTail = line.getTail();
    System.out.println(pointTail);
    pointHead = line.getHead();
    System.out.println(pointHead);

} //end main
} //end controlling class PointLine02

```

Listing 16 . Source code for the game-programming math library named GM2D01.

/*GM2D01.java
Copyright 2008, R.G.Baldwin

The name GM2D01 is an abbreviation for
GameMath2D01.

This is a game-math class, which will be expanded
over
time. The class is provided solely for educational
purposes. No effort has been expended to optimize
it in
any way. Rather, it was designed and implemented
for
maximum clarity in order to help students
understand
the programming details of various mathematical
operations
commonly used in game programming.

Each time the class is expanded or modified, it
will be
given a new name by incrementing the two digits at
the
end of the name. No attempt will be made to
maintain
backward compatibility from one version of the
class to
the next.

This class contains a number of static top-level
classes.
This organizational approach was used primarily
for the
purpose of gathering such classes under a single
naming
umbrella while avoiding name conflicts within a
single

package. For example, as time passes, and this library is expanded, my default package may contain class files with the following names:

```
GM2D01$Point.class  
GM2D02$Point.class  
GM2D03$Point.class
```

All real-number values used in this class are maintained as type double.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/
```

```
public class GM2D01{  
  
    //An object of this class represents a 2D column  
    matrix.  
    // An object of this class is the fundamental  
    building  
    // block for several of the other classes in the  
    // library.  
    public static class ColMatrix{  
        double[] data = new double[2];  
  
        ColMatrix(double data0,double data1){  
            data[0] = data0;  
            data[1] = data1;  
        }//end constructor  
  
        public String toString(){  
            return data[0] + "," + data[1];  
        }//end overridden toString method
```

```

//end constructor setting method

    public double getData(int index){
        if((index < 0) || (index > 1))
            throw new
IndexOutOfBoundsException();
        return data[index];
    }//end getData

} //end class ColMatrix

//=====
====//

    public static class Point{
        GM2D01.ColMatrix point;

        Point(GM2D01.ColMatrix point){
            this.point = point;
        }//end constructor

        public String toString(){
            return point.getData(0) + "," +
point.getData(1);
        }//end toString

        public double getData(int index){
            if((index < 0) || (index > 1))
                throw new
IndexOutOfBoundsException();
            return point.getData(index);
        }//end getData

    } //end class Point

//=====
====//

```

```

public static class Vector{
    GM2D01.ColMatrix vector;

    Vector(GM2D01.ColMatrix vector){
        this.vector = vector;
    }//end constructor

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1);
    }//end toString

    public double getData(int index){
        if((index < 0) || (index > 1))
            throw new
IndexOutOfBoundsException();
        return vector.getData(index);
    }//end getData

} //end class Vector

//=====
====//

//A line is defined by two points. One is called
the
// tail and the other is called the head.
public static class Line{
    GM2D01.Point[] line = new GM2D01.Point[2];

    Line(GM2D01.Point tail, GM2D01.Point head){
        this.line[0] = tail;
        this.line[1] = head;
    }//end constructor

    public String toString(){
        return "Tail = " + line[0].getData(0) + "."

```

```

        + line[0].getData(1) + "\nHead = "
        + line[1].getData(0) + ","
        + line[1].getData(1);
    }//end toString

    public GM2D01.Point getTail(){
        return line[0];
    }//end getTail

    public GM2D01.Point getHead(){
        return line[1];
    }//end getTail

} //end class Line

} //end class GM2D01

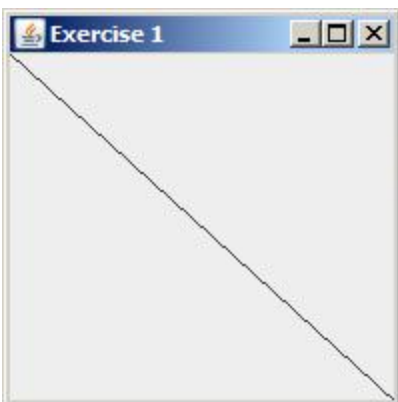
```

Exercises

Exercise 1

Without using the game math library, use the programming environment of your choice to write a program that draws a diagonal line from the upper-left to the lower-right corner of a window as shown in [Figure 7](#).

Figure 7 Graphic output from Exercise 1.



Exercise 2

Using Java and the game math library named **GM2D01** , write a program that:

- Creates a new **GM2D01.Point** object with coordinate values of 3.4 and 9.7.
- Uses the overridden **toString** method to get and display the location of the point in 2D space in the format shown by the first line of text in [Figure 8](#).
- Uses the **getData** method to get and display the location of the point in 2D space in the format shown by the last two lines of text in [Figure 8](#).

Your screen output should display numeric values in the format shown in [Figure 8](#) where each ? character represents a single digit.

Figure 8 . Text output from Exercise 2.

```
? . ? , ? . ?  
? . ?  
? . ?
```

Exercise 3

Using Java and the game math library named **GM2D01** , write a program that:

- Represents a line segment using a **GM2D01.Line** object with the ends of the line segment being located at the following coordinates:
 - x=2.2, y=5.3
 - x=5.2, y=9.3
- Displays the information shown in [Figure 9](#) to describe the line segment.

Your screen output should display numeric values in the format shown in [Figure 9](#) where each ? character represents a single digit. *(Note that the number of digits to the right of the decimal point in the last line of text may be greater or less than that shown in [Figure 9](#).)*

Figure 9 . Text output from Exercise 3.

```
Tail = ?.?,?.?  
Head = ?.?,?.?  
Length = ?.???
```

-end-

Game0105r Review

This module contains review questions and answers keyed to the module titled GAME 2302-0105: Getting Started.

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [30](#), [31](#), [32](#), [33](#), [34](#), [35](#), [36](#), [37](#), [38](#), [39](#), [40](#), [41](#), [42](#), [43](#), [44](#), [45](#), [46](#), [47](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module contains review questions and answers keyed to the module titled [GAME 2302-0105: Getting Started](#).

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

Successful game programmers don't need good math skills.

True or False?

[Answer 1](#)

Question 2

This set of modules relies strictly on procedural programming.

[Answer 2](#)

Question 3

You will need skills in the mathematical areas that are required for success in 2D and 3D graphics programming.

True or False?

[Answer 3](#)

Question 4

As a minimum, you will need skills in:

Select the correct answers

1. Geometry
2. Trigonometry
3. Vectors
4. Matrices
5. 2D and 3D transforms
6. Transformations between coordinate systems
7. Projections
8. All of the above

[Answer 4](#)

Question 5

A column matrix can be used to represent a point in a given coordinate frame.

True or False?

[Answer 5](#)

Question 6

When representing a point with a column matrix, the same point will have the same representation in different coordinate frames.

True or False?

[Answer 6](#)

Question 7

You must use an IDE such as Eclipse or NetBeans to create Java source code.

True or False?

[Answer 7](#)

Question 8

There is no relationship between the name of the source code file and the name of the class defined in the file.

True or False?

[Answer 8](#)

Question 9

The name of the Java compiler program is java.exe.

True or False?

[Answer 9](#)

Question 10

The class named ***Point2D.Double*** is a class in the standard Java library.

True or False?

[Answer 10](#)

Question 11

The following is the correct syntax for the **main** method in a Java application.

Note:

```
public static void main(){  
    GUI guiObj = new GUI();  
} //end main
```

True or False?

[Answer 11](#)

Question 12

The following Java code instantiates a new object of a class named **GUI** and saves that object's reference in the variable named **guiObj**.

Note:

```
class PointLine01{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class PointLine01
```

[Answer 12](#)

Question 13

The class named **GUI** is a class in the Java 7 Standard Edition library.

True or False?

[Answer 13](#)

Question 14

The name of the constructor for a Java class must match the name of the class.

True or False?

[Answer 14](#)

Question 15

The method named **setDefaultCloseOperation** can be called on a Java **JFrame** object to establish the behavior of the button with the X in the upper-right corner of the visual manifestation of the **JFrame** object.

True or False?

[Answer 15](#)

Question 16

A **Canvas** object's overridden **paint** method will be executed whenever the **JFrame** containing the **Canvas** is displayed on the computer screen, whenever that portion of the screen needs to be repainted, or whenever the **repaint** method is called on the canvas.

True or False?

[Answer 16](#)

Question 17

A **Canvas** object's overridden **paint** method always receives an incoming parameter as type **Button** .

True or False?

[Answer 17](#)

Question 18

The class named **Graphics2D** is a superclass of the class named **Graphics** .

True or False?

[Answer 18](#)

Question 19

Translating the origin from the default position at the upper-left corner of a **Canvas** object to the center of the **Canvas** object is equivalent to creating a new *coordinate frame* as explained by Kjell.

True or False?

[Answer 19](#)

Question 20

The default direction for increasing vertical coordinate values in a **Canvas** object is down the screen.

True or False?

[Answer 20](#)

Question 21

The default direction for increasing horizontal coordinate values in a **Canvas** object is from right to left across the screen.

True or False?

[Answer 21](#)

Question 22

According to Kjell, a line drawn on the screen using an object of the **Line2D.Double** class is a true line segment.

True or False?

[Answer 22](#)

Question 23

Objects instantiated from the classes named **Point2D.Double** and **Line2D.Double** are well suited for inclusion in mathematical operations.

True or False?

[Answer 23](#)

Question 24

Creation of 2D and 3D graphics requires that you always display the points.

True or False?

[Answer 24](#)

Question 25

A vector is a geometrical object that has two properties: length and direction." He also tells us, "A vector does not have a position.

True or False?

[Answer 25](#)

Question 26

A vector must always specify a position.

True or False?

[Answer 26](#)

Question 27

Three real numbers are required to represent a vector in a 2D system.

True or False?

[Answer 27](#)

Question 28

A column matrix can be used to represent a point and can also be used to represent a vector.

True or False?

[Answer 28](#)

Question 29

Different column matrices can be used to represent the same vector in different reference frames, in which case, the contents of the matrices will be different.

True or False?

[Answer 29](#)

Question 30

The two (*or three*) real number values contained in a column matrix to represent a point specify an absolute location in space relative to the current coordinate frame.

True or False?

[Answer 30](#)

Question 31

The two (*or three*) real number values contained in a column matrix to represent a vector (*in 2D or 3D*) specify a *displacement* of a specific distance from an arbitrary point in an arbitrary direction.

True or False?

[Answer 31](#)

Question 32

With regard to a vector, in 2D, the two values contained in a column matrix represent the displacements along three orthogonal axes.

True or False?

[Answer 32](#)

Question 33

In the case of 2D, the length of the vector is the length of the hypotenuse of a right triangle formed by the x and y displacement values.

True or False?

[Answer 33](#)

Question 34

In 2D, the direction of a vector can be determined from the angle formed by the x-displacement and the line segment that represents the hypotenuse of a right triangle formed by the x, y, and z displacements.

True or False?

[Answer 34](#)

Question 35

A point is an *absolute location* , but a vector is a *displacement* .

True or False?

[Answer 35](#)

Question 36

This program instantiates objects from the following non-static top-level classes belonging to the class named **GM2D01** :

- GM2D01.ColMatrix
- GM2D01.Line
- GM2D01.Point
- GM2D01.Vector

True or False?

[Answer 36](#)

Question 37

Type **int** is the default representation for literal real numbers in Java.

True or False?

[Answer 37](#)

Question 38

*When an object's reference is passed as a parameter to the **System.out.println** method, the **getClass** method belonging to the object is executed automatically.*

True or False?

[Answer 38](#)

Question 39

The **toString** method is overloaded in the **ColMatrix** class.

True or False?

[Answer 39](#)

Question 40

If you call the **getData** method belonging to an object of the **GM2D01.ColMatrix** class and pass a parameter value of 2, the program

will throw an **IndexOutOfBoundsException** .

True or False?

[Answer 40](#)

Question 41

The constructor for the **GM2D01.ColMatrix** class requires three incoming parameters of type **double** .

True or False?

[Answer 41](#)

Question 42

The constructor for the **GM2D01.Point** class requires a single incoming parameter, which is a reference to an object of the class **GM2D01.ColMatrix** .

True or False?

[Answer 42](#)

Question 43

GM2D01.Vector and **java.util.Vector** are simply different names for the same class.

True or False?

[Answer 43](#)

Question 44

The constructor for the **GM2D01.Vector** class requires two incoming parameters, which are references to objects of the class **GM2D01.ColMatrix** .

True or False?

[Answer 44](#)

Question 45

A line segment is the straight path between two points. It has no thickness and therefore cannot be seen by the human eye.

True or False?

[Answer 45](#)

Question 46

The constructor for the **GM2D01.Line** class requires two incoming parameters, which are references to objects of the class **GM2D01.ColMatrix** .

True or False?

[Answer 46](#)

Question 47

The **GM2D01** library provides methods for rendering objects of the **ColMatrix** , **Line** , **Point** , or **Vector** classes in a visual graphics context.

True or False?

[Answer 47](#)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 47

False. The library named **GM2D01** is purely mathematical. The library doesn't provide any mechanism for rendering objects of the **ColMatrix** , **Line** , **Point** , or **Vector** classes in a visual graphics context.

[Back to Question 47](#)

Answer 46

False. The constructor for the **GM2D01.Line** class requires two incoming parameters, which are references to objects of the class **GM2D01.Point** .

[Back to Question 46](#)

Answer 45

True

[Back to Question 45](#)

Answer 44

False. The constructor for the **GM2D01.Vector** class requires a single incoming parameter, which is a reference to an object of the class **GM2D01.ColMatrix** .

[Back to Question 44](#)

Answer 43

False. **GM2D01.Vector** is a different class from the class named **java.util.Vector** in the standard Java library.

[Back to Question 43](#)

Answer 42

True

[Back to Question 42](#)

Answer 41

False. The constructor for the **GM2D01.ColMatrix** class requires two incoming parameters of type **double** .

[Back to Question 41](#)

Answer 40

True

[Back to Question 40](#)

Answer 39

False. The **toString** method is *overridden* in the **ColMatrix** class.

[Back to Question 39](#)

Answer 38

False. *When an object's reference is passed as a parameter to the **System.out.println** method, the **toString** method belonging to the object is executed automatically.*

[Back to Question 38](#)

Answer 37

False. Type **double** is the default representation for literal real numbers in Java.

[Back to Question 37](#)

Answer 36

False.

This program instantiates objects from the following *static* top-level classes belonging to the class named **GM2D01** :

- GM2D01.ColMatrix
- GM2D01.Line
- GM2D01.Point
- GM2D01.Vector

[Back to Question 36](#)

Answer 35

True

[Back to Question 35](#)

Answer 34

False. In 2D, the direction of a vector can be determined from the angle formed by the x-displacement and the line segment that represents the hypotenuse of a right triangle formed *by the x and y* displacements.

[Back to Question 34](#)

Answer 33

True

[Back to Question 33](#)

Answer 32

False. With regard to a vector, in 2D, the two values contained in a column matrix represent the displacements along *a pair* of orthogonal axes.

[Back to Question 32](#)

Answer 31

False. Kjell tells us that the two (*or three*) real number values contained in the matrix to represent a vector (*in 2D or 3D*) specify a *displacement* of a specific distance from an arbitrary point in a *specific* direction.

[Back to Question 31](#)

Answer 30

True

[Back to Question 30](#)

Answer 29

True

[Back to Question 29](#)

Answer 28

True

[Back to Question 28](#)

Answer 27

False. Kjell tells us that we can represent a vector with two real numbers in a 2D system and with three real numbers in a 3D system.

[Back to Question 27](#)

Answer 26

False. According to Kjell, *"A vector does not have a position."*

[Back to Question 26](#)

Answer 25

True. According to Kjell, *"A vector is a geometrical object that has two properties: length and direction."* He also tells us, *"A vector does not have a position."*

[Back to Question 25](#)

Answer 24

False. There will be many occasions when you, as a game programmer, will need to define the coordinate values for a point (*or a set of points*) that you have no intention of displaying on the screen. Instead, you will use those points for various mathematical operations to produce something else that may or may not be displayed on the screen.

[Back to Question 24](#)

Answer 23

False. Objects instantiated from the classes named **Point2D.Double** and **Line2D.Double** are intended primarily for rendering graphics on the screen and are **not** well suited for inclusion in mathematical operations. That is part of the rationale behind the development of the game-math library: *separation of data objects that are suitable for mathematical operations from graphics objects* .

[Back to Question 23](#)

Answer 22

False. According to Kjell, the true line segment has no width, and therefore is not visible to the human eye. This is typically not the case with a line drawn on the screen using an object of the **Line2D.Double** class.

[Back to Question 22](#)

Answer 21

False. The default direction for increasing horizontal coordinate values in a **Canvas** object is from *left to right* across the screen.

[Back to Question 21](#)

Answer 20

True

[Back to Question 20](#)

Answer 19

True

[Back to Question 19](#)

Answer 18

False. The class named **Graphics2D** is a *subclass* of (*inherits from or is derived from*) the class named **Graphics** .

[Back to Question 18](#)

Answer 17

False. A **Canvas** object's overridden **paint** method always receives an incoming parameter as type **Graphics** .

[Back to Question 17](#)

Answer 16

True

[Back to Question 16](#)

Answer 15

True

[Back to Question 15](#)

Answer 14

True

[Back to Question 14](#)

Answer 13

False. There is no class named **GUI** in the Java 7 Standard Edition library. It is a name that I often use for custom classes that present a Graphical User Interface.

[Back to Question 13](#)

Answer 12

True

[Back to Question 12](#)

Answer 11

False. The correct syntax is shown below. Note the formal argument list of the **main** method.

Note:

```
public static void main(String[] args){  
    GUI guiObj = new GUI();  
} //end main
```

[Back to Question 11](#)

Answer 10

True

[Back to Question 10](#)

Answer 9

False. The name of the Java compiler program is **javac.exe** . The file named **java.exe** is the Java virtual machine.

[Back to Question 9](#)

Answer 8

False. The class having the **main** method must be contained in a file having the same name as the class with an extension of .java. Good programming practice dictates that even for other classes, the name of each source code file should match the name of the Java class defined in the file.

[Back to Question 8](#)

Answer 7

False. You can use just about any text editor to create your Java source code files as text files with an extension of .java.

[Back to Question 7](#)

Answer 6

False. The same point will have different representations in different coordinate frames.

[Back to Question 6](#)

Answer 5

True

[Back to Question 5](#)

Answer 4

8. All of the above

[Back to Question 4](#)

Answer 3

True

[Back to Question 3](#)

Answer 2

False. In addition to helping you with your math skills, I will also teach you how to incorporate those skills into object-oriented programming using Java.

[Back to Question 2](#)

Answer 1

False. In order to be a successful game programmer you must be skilled in technologies other than simply programming. Those technologies include mathematics.

[Back to Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0105r Review: Getting Started
- File: Game0105r.htm
- Published: 12/31/12
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available

on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0110: Updating the Game Math Library for Graphics

Learn how to update the game-math library to provide new capabilities including the addition of graphics and set methods for column matrices, points, vectors, and lines. Also study sample programs that illustrate the use of the new capabilities and learn how to draw on off-screen images.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Preview](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [The game-programming library named GM2D02](#)
 - [The program named PointLine03](#)
 - [The program named PointLine04](#)
- [Documentation for the GM2D02 library.](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary.](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)
 - [Exercise 3](#)
 - [Exercise 4](#)
 - [Exercise 5](#)
 - [Exercise 6](#)
 - [Exercise 7](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in

Austin, TX.

What you have learned

In the previous module, I presented and explained two sample programs and a sample game-programming math library named **GM2D01** , which were intended to implement concepts from Kjell's tutorial in Java code.

The library named **GM2D01** is purely mathematical. By that, I mean that the library doesn't provide any mechanism for rendering objects of the **ColMatrix** , **Line** , **Point** , or **Vector** classes in a visual graphics context. That capability will be added to the version that I will present in this module.

What you will learn

In this module, you will learn how to update the math library to provide a number of new capabilities including the addition of graphics and various *set* methods. You will also learn that the addition of *set* methods exposes certain vulnerabilities, and you will learn how to protect against those vulnerabilities. Finally, you will learn how to draw on off-screen images.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output from both sample programs.
- [Figure 2](#). Graphic output from Exercise 1.
- [Figure 3](#). Graphic output from Exercise 2.
- [Figure 4](#). Graphic output from Exercise 3.
- [Figure 5](#). Text output from Exercise 3.
- [Figure 6](#). Graphic output from Exercise 4.
- [Figure 7](#). Graphic output from Exercise 5.
- [Figure 8](#). Graphic output from Exercise 6.
- [Figure 9](#). Graphic output from Exercise 7.

Listings

- [Listing 1](#). Beginning of the class named PointLine03 and the class named GUI.
- [Listing 2](#). Beginning of the constructor for the GUI class.
- [Listing 3](#). Create two off-screen images.
- [Listing 4](#). Remaining code in the constructor.
- [Listing 5](#). Beginning of the drawOffscreen method.
- [Listing 6](#). Define four lines that will be used to draw borders.
- [Listing 7](#). Draw a visual manifestation of each line.
- [Listing 8](#). The draw method of the GM2D02.Line class.
- [Listing 9](#). Draw a visual manifestation of a GM2D02.Point object.
- [Listing 10](#). The draw method of the GM2D02.Point class.
- [Listing 11](#). Draw the vertices of a hexagon.
- [Listing 12](#). Draw six lines connecting the vertices of the hexagon.
- [Listing 13](#). Instantiate three objects of type GM2D02.Vector.
- [Listing 14](#). Call the draw method to draw the vector.
- [Listing 15](#). The draw method of the GM2D02.Vector class.
- [Listing 16](#). Three visual manifestations of the same vector.
- [Listing 17](#). Draw the blue vector.
- [Listing 18](#). The MyCanvas class and the overridden paint method.
- [Listing 19](#). The setData method of the ColMatrix class.
- [Listing 20](#). Updated constructor for the Point class in GM2D02.
- [Listing 21](#). The drawOffscreen method of the program named PointLine04.
- [Listing 22](#). Instantiate a point at the upper-left corner.
- [Listing 23](#). Instantiate a moveable Point object.
- [Listing 24](#). Instantiate a moveable Line object.
- [Listing 25](#). Relocate the Line to three more locations.
- [Listing 26](#). Source code for the game-math library class named GM2D02.
- [Listing 27](#). Source code for the program named PointLine03.
- [Listing 28](#). Source code for the program named PointLine04.
- [Listing 29](#). Code for generating random values in Java.

General background information

So far, we have been dealing with column matrices, points, vectors, and lines. (*We will get to other topics in future modules.*)

What is a point?

According to Kjell, a point is simply a location in space. It has no width, depth, or height. Therefore, it cannot be seen by the human eye, which means that we can't draw a point on the computer screen. However, it is possible to draw an object on the computer screen that indicates the location of the point.

What do I mean by this?

Suppose you go out onto your driveway and establish that one corner of the driveway represents the origin in a rectangular Cartesian coordinate framework. Then you use measuring devices to identify a specific location on the driveway. You can point to that location with your finger, and others can see your finger, but they can't see the point. Once again, the point has no width, height, or depth, and therefore cannot be seen by the human eye.

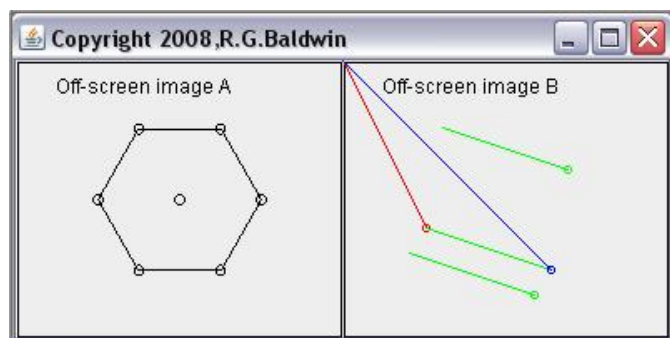
Mark the point

What you can do, however, is to take a piece of chalk and draw a small circle around your finger. Then others can see the mark that you have drawn as an estimate of the location of the point. It is very important however to remember that the chalk mark is not the point. It is simply a visual object that indicates the location of the point to some degree of accuracy. The actual location of the point is a piece of *underlying data*. The chalk mark is a graphic object that you have invented to represent that underlying data in a visual way to a human observer.

Program output

Both of the sample programs that I will explain in this module will produce the same graphic screen output, which is shown in [Figure 1](#).

Figure 1 Screen output from both sample programs.



The left image in [Figure 1](#) contains seven small circles. Each of those seven circles marks the location of a point in the 2D space of the image. However, those circles are not the points. The seven *points* consist solely of coordinate values and are not visible to the human eye. The seven circles are graphics objects that were placed there to mark the locations in space of the seven points. The points existed in the 2D space before the graphics objects were created, and would have existed even if I had not caused the circles to be drawn to mark the locations of the points.

What is a line segment?

This can be even more confusing. Kjell tells us that a line segment is the straight path between two points, and that it has no thickness. Since it has no thickness, a line cannot be

seen by the human eye.

The left image in [Figure 1](#) contains six graphic objects that we are likely to call lines or line segments. True enough, those graphic objects mark the path taken by the *lines* that represent straight paths between the pairs of points whose locations are indicated by the circles. However, those graphic objects are not *lines* in the context of this discussion. Rather, they are simply graphic objects that were used to mark the paths of the lines in a visual way for the benefit of a human observer.

Why am I emphasizing this so heavily?

Shortly, you will see the changes that I made to the game-math library in preparation for this module. That library makes it possible for you to easily create and manipulate underlying data objects for column matrices, points, vectors, and lines. Those changes include the ability for points, lines, and vectors to create graphical objects that represent themselves and to display those graphical objects on a specified graphics context upon request.

The left image in [Figure 1](#) shows the manner in which **GM2D02.Point** objects and **GM2D02.Line** objects represent themselves on a graphics context.

The right image in [Figure 1](#) shows the manner in which **GM2D02.Vector** objects represent themselves on a graphics context.

It is very important to understand the difference between the objects that encapsulate the underlying data and the graphic objects that represent those objects for visible human consumption. Later on, we will be doing math using the underlying data objects. However, it is generally not possible to do math using the graphic objects shown in [Figure 1](#).

Additional modifications that I made to the library for this module makes it is easy to modify the values encapsulated in one of the underlying data objects. However, once the graphical representation of one of those data objects is drawn on the graphics context, it is fairly difficult to modify, particularly if it overlaps another graphic object that you don't want to modify. The library does not provide that capability.

All Java parameters are passed to methods by value

Moving to a completely different topic, when you call out the name of a variable as a parameter to be passed to a method in Java, a copy of the contents of that variable is made and the copy is passed to the method. Code in the method can modify the copy but it cannot modify the contents of the original variable. This is typically referred to as passing parameters *by value*. (*Despite what you may have read elsewhere, unlike C and C++, Java parameters are never passed by reference.*)

However, Java supports both primitive variables and reference variables. Passing reference variables by value exposes some vulnerabilities in the original version of the library when

set methods are added to the library. A complete explanation of this topic is beyond the scope of this module. However, I will show you how I modified the code in the library to protect against such vulnerabilities.

Preview

I will explain the changes to the library in this module in conjunction with two programs that illustrate the impact of those changes. Both programs produce the screen output shown in [Figure 1](#), but they do so in significantly different ways.

The primary topics to be discussed have to do with:

- Making it possible for the underlying data objects to represent themselves by drawing graphical objects on a specified 2D graphics context upon request.
- Making it possible to call **set** methods to modify the data encapsulated in the underlying data objects.
- Protecting the data from corruption due to certain vulnerabilities exposed by adding **set** methods to the library.

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

Much of the code in the library remains unchanged. I explained that code in the previous module and I won't repeat that explanation in this module. Rather, in this module, I will concentrate on explaining the modifications that I made to the library.

The game-programming library named **GM2D02**

A complete listing of the modified game-programming library named **GM2D02** is provided in [Listing 26](#).

This library is an update of the earlier game-math library named **GM2D01**. This update added the following new capabilities:

- Draw circles to represent **GM2D02.Point** objects.
- Draw visible lines to represent **GM2D02.Line** objects.
- Draw visible lines and circles to represent **GM2D02.Vector** objects.
- Call a **setData** method to change the values in a **GM2D02.ColMatrix** object.
- Call a **setData** method to change the values in a **GM2D02.Point** object.
- Call a **setData** method to change the values in a **GM2D02.Vector** object.

- Call a **setHead** method or a **setTail** method to change one of the points that defines a **GM2D02.Line** object.

Constructors were updated to ensure that existing points, vectors, and lines are not corrupted by using the new *set* methods to change the values in the **ColMatrix** and/or **Point** objects originally used to construct the points, vectors, and lines. The updated constructors create and save clones of the **ColMatrix** and/or **Point** objects originally used to define the **Point** , **Vector** , and/or **Line** objects.

I will explain the new code in the library in conjunction with the explanations of the programs named **PointLine03** and **PointLine04** .

The program named **PointLine03**

A complete listing of the program named **PointLine03** is provided in [Listing 27](#) .

This program illustrates the use of *draw* methods that were added to the **GM2D02** game-math library to produce visual manifestations of **Point** , **Line** , and **Vector** objects instantiated from classes in the game-math library. The program also illustrates drawing on off-screen images and then copying those images to a **Canvas** object in an overridden **paint** method.

As mentioned earlier, this program produces the screen output shown in [Figure 1](#) .

Beginning of the class named **PointLine03** and the class named **GUI**

[Listing 1](#) shows the beginning of the class named **PointLine03** (*including the **main** method*) and the beginning of the class named **GUI** .

Listing 1 . Beginning of the class named **PointLine03 and the class named **GUI**.**

Listing 1 . Beginning of the class named PointLine03 and the class named GUI.

```
class PointLine03{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class PointLine03
//=====================================================//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a JFrame
    // object.
    int hSize = 400;
    int vSize = 200;
    Image osiA;//one off-screen image
    Image osiB;//another off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
```

The code in the **main** method instantiates an object of the **GUI** class. The code showing in the **GUI** class definition declares several instance variables, initializing some of them.

Beginning of the constructor for the GUI class

[Listing 2](#) shows the beginning of the constructor for the **GUI** class.

Listing 2 . Beginning of the constructor for the GUI class.

Listing 2 . Beginning of the constructor for the GUI class.

```
GUI(){//constructor
    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create a new drawing canvas and add it to the
    // center of the JFrame.
    myCanvas = new MyCanvas();
    this.getContentPane().add(myCanvas);

    //This object must be visible before you can get an
    // off-screen image. It must also be visible before
    // you can compute the size of the canvas.
    setVisible(true);
    osiWidth = myCanvas.getWidth()/2;
    osiHeight = myCanvas.getHeight();
```

The code in [Listing 2](#) is straightforward and shouldn't require an explanation beyond the embedded comments.

Create two off-screen images

Here is where things begin to get interesting. [Listing 3](#) creates two off-screen images and gets a graphics context on each of them.

Listing 3 . Create two off-screen images.

```
osiA = createImage(osiWidth,osiHeight);
Graphics2D g2Da = (Graphics2D)(osiA.getGraphics());

osiB = createImage(osiWidth,osiHeight);
Graphics2D g2Db = (Graphics2D)(osiB.getGraphics());
```

Java programs are not constrained to simply draw on the screen. It is possible to create an off-screen image, get a graphics context on it, and use methods of the **Graphics** and/or **Graphics2D** classes to draw on the off-screen image.

The off-screen image can then be used for a variety of purposes including:

- writing the image into a disk file.
- copying the image to a **Canvas** object for display on the screen.

Will copy off-screen images to the screen

In this program, we will copy each of the off-screen images to a **Canvas** object in a side-by-side format (see [Figure 1](#)) and display the **Canvas** object on the content pane in a **JFrame** object.

While animation is not the purpose of using off-screen images in this program, such off-screen images are often used to produce clean animation. Code is executed to produce an off-screen image and then the entire image is copied to the screen in a single blast. This prevents the user from seeing the development of the image that might otherwise be visible if the code was executed to produce the image directly onto the screen.

The `createImage` method

There are seven different versions of the **createImage** method in the JDK 1.7 standard library. The version of the method used in this program is defined in the **Component** class and inherited into the **JFrame** class. According to the Sun documentation, this method creates an off-screen drawable image to be used for double buffering.

The two required parameters are width and height. If the image is later displayed on the screen, the units of width and height are measured in pixels. The values contained in the width and height parameters in [Listing 3](#) were computed in [Listing 2](#). The width of each off-screen image is set to half the width of the canvas. The height of each off-screen image is set to the height of the canvas. This facilitates the display of the two images in the side-by-side format shown in [Figure 1](#).

Also, according to the Sun documentation, the method returns *"an off-screen drawable image, which can be used for double buffering. The return value may be null if the component is not displayable."* The returned value is type **Image**.

The `getGraphics` method

There are eleven different versions of the **getGraphics** method in the JDK 1.7 standard library. The version of the method used in this program is defined in the **Image** class. This method is called on the **Image** object returned by the **createImage** method in [Listing 3](#).

According to the Sun documentation, this method creates a graphics context for drawing to an off-screen image. You will note that the reference to the graphics context was cast to type **Graphics2D** and saved in a reference variable of the type **Graphics2D** . I explained the relationship between the **Graphics** class and the **Graphics2D** class in an earlier module.

At this point, we have two off-screen image objects of type **Image** and we have a graphics context on each of them. This makes it possible to use the methods of the **Graphics** class and/or the methods of the **Graphics2D** class to draw on either of the off-screen images.

Remaining code in the constructor

[Listing 4](#) shows the remainder of the constructor for the **GUI** class.

Listing 4 . Remaining code in the constructor.

```
drawOffscreen(g2Da, g2Db);  
  
myCanvas.repaint();  
  
} //end constructor
```

[Listing 4](#) begins by calling the method named **drawOffscreen** to draw some points, lines, and vectors on the two off-screen images. Note that references to each of the two graphics contexts are passed as parameters to the **drawOffscreen** method.

Then [Listing 4](#) calls the **repaint** method belonging to the **MyCanvas** object to cause the overridden **paint** method belonging to that object to be executed. Later, we will see that the overridden **paint** method copies the two off-screen images to the canvas in the side-by-side format shown in [Figure 1](#) causing the contents of those off-screen images to be displayed in a **JFrame** object.

Beginning of the drawOffscreen method

The beginning of the **drawOffscreen** method is shown in [Listing 5](#) . The purpose of this method is to define points, lines, and vectors as underlying data objects and then to cause a visual manifestation of some of the points, lines, and vectors to be drawn onto the two off-screen images.

Listing 5 . Beginning of the drawOffscreen method.

```
void drawOffscreen(Graphics2D g2Da,Graphics2D g2Db){  
  
    //Draw a label on each off-screen image.  
    g2Da.drawString("Off-screen image A",  
                    osiWidth/8,  
                    osiHeight/8);  
    g2Db.drawString("Off-screen image B",  
                    osiWidth/8,  
                    osiHeight/8);  
}
```

Before getting into the underlying data objects, however, [Listing 5](#) calls the **drawString** method belonging to the **Graphics** class to draw the two text strings shown in [Figure 1](#) on the two off-screen images.

Instantiate four Line objects that will be used to draw borders

[Listing 6](#) instantiates four **GM2D02.Line** objects that will ultimately be used to draw the borders around the left and right images shown in [Figure 1](#).

Listing 6 . Define four lines that will be used to draw borders.

Listing 6 . Define four lines that will be used to draw borders.

```
//First define four points that will be used to
define
// the ends of the four lines.
GM2D02.Point upperLeftPoint = new GM2D02.Point(
    new
GM2D02.ColMatrix(1.0,1.0));
GM2D02.Point upperRightPoint = new GM2D02.Point(
    new GM2D02.ColMatrix(osiWidth-
1,1.0));
GM2D02.Point lowerRightPoint = new GM2D02.Point(
    new GM2D02.ColMatrix(osiWidth-1,osiHeight-
1));
GM2D02.Point lowerLeftPoint = new GM2D02.Point(
    new GM2D02.ColMatrix(1.0,osiHeight-
1));

//Now define the four lines based on the endpoints.
GM2D02.Line top = new GM2D02.Line(upperLeftPoint,
    upperRightPoint);
GM2D02.Line rightSide = new GM2D02.Line(
    upperRightPoint,
lowerRightPoint);
GM2D02.Line bottom = new GM2D02.Line(lowerLeftPoint,
lowerRightPoint);
GM2D02.Line leftSide = new
GM2D02.Line(upperLeftPoint,
lowerLeftPoint);
```

Nothing new here

There is nothing in [Listing 6](#) that is new relative to what you learned in the previous module. [Listing 6](#) starts by defining four **GM2D02.Point** objects that will be used to define the ends of the line segments. References to these **Point** objects are passed to the constructor for the **GM2D02.Line** class to instantiate the four **Line** objects.

Note that the locations of the points, and hence the positions of the lines were one pixel away from the edges of the canvas towards the center of the canvas. These locations were

chosen to ensure that the resulting visual manifestations of the lines would be visible on the canvas.

It is also extremely important to understand that the **GM2D02.Point** and **GM2D02.Line** objects instantiated in [Listing 6](#) are not graphical objects. Rather, they are underlying data objects, which are suitable for use in mathematical operations.

Draw a visual manifestation of each line

[Listing 7](#) calls the **draw** method of the **GM2D02.Line** class four times in succession for each off-screen image. Note that a reference to the off-screen image is passed as a parameter to the **draw** method each time the method is called on one of the **GM2D02.Line** objects.

Listing 7 . Draw a visual manifestation of each line.

```
//Now draw a visual manifestation of each line
// on g2Da.
top.draw(g2Da);
rightSide.draw(g2Da);
bottom.draw(g2Da);
leftSide.draw(g2Da);

//Now draw a visual manifestation of each of the same
// four lines on g2Db
top.draw(g2Db);
rightSide.draw(g2Db);
bottom.draw(g2Db);
leftSide.draw(g2Db);
```

As you will see shortly, each of these calls to the **draw** method causes an object of the standard Java **Line2D.Double** class to be rendered onto the specified off-screen image. When the off-screen image is ultimately copied to the canvas object by the overridden **paint** method, this produces a visual manifestation of the **GM2D02.Line** object.

Once again, however, it is very important to understand that the **Line2D.Double** object is a graphical object and the **GM2D02.Line** is an underlying data object. They are completely independent objects.

Also, it is important to understand that the **Line2D.Double** class is a member of the standard Java library, whereas the **GM2D02.Line** class is a member of my special game math library named **GM2D02** .

The draw method of the GM2D02.Line class

Now it's time to explain one of the methods that was added to the original game-math library named **GM2D01** to produce the new library named **GM2D02** . At this point, I will start switching back and forth between code in the **GM2D02** library and code in the program named **PointLine03** .

[Listing 8](#) shows the **draw** method that was called repeatedly on the **GM2D02.Line** objects in [Listing 7](#). This code belongs to the **GM2D02** library.

Listing 8 . The draw method of the GM2D02.Line class.

```
public void draw(Graphics2D g2D){
    Line2D.Double line = new Line2D.Double(
                                getTail().getData(0),
                                getTail().getData(1),
                                getHead().getData(0),
                                getHead().getData(1));
    g2D.draw(line);
} //end draw
```

Construct and render a Line2D.Double object on the graphics context

[Listing 8](#) calls the **getTail** and **getHead** methods to get the x and y coordinate values (*relative to the current coordinate frame*) that define the ends of the line segment represented by the **GM2D02.Line** object on which the **draw** method was called.

These four values are passed as parameters to the constructor for the Java standard **Line2D.Double** class to define the endpoints of a **Line2D.Double** object.

Then the **draw** method of the **Graphics2D** class is called on the reference to the incoming graphics context parameter to cause the **Line2D.Double** object to be rendered onto that graphics context.

In this case, that graphics context is an off-screen graphics context. However, it could just as easily be an on-screen graphics context such as the graphics context that is received as a parameter to an overridden **paint** method.

Note: Two different draw methods are involved:

The code in my custom **draw** method, which is a member of my custom **GM2D02.Line** class, turns around and calls a completely different method named **draw**, which is a member of the standard Java **Graphics2D** class.

Hallmarks of object-oriented programming

An object of the **GM2D02.Line** class knows how to produce a visual manifestation of itself onto a specified graphics context as illustrated by the borders surrounding the left and right images in [Figure 1](#).

This is one of the hallmarks of object-oriented programming. Objects know how to do useful things for themselves. Later on, we will see other hallmarks such as the ability of an object of the **GM2D02.Vector** class to add itself to another object of the same class and return a new vector object that is the sum of the two vectors.

Draw a visual manifestation of a GM2D02.Point object

Returning to the **drawOffScreen** method of the program named **PointLine03**, [Listing 9](#) begins by calling the **translate** method of the **Graphics** class on the off-screen image shown in the left side of [Figure 1](#) for the purpose of changing the origin from the upper-left corner of the graphics context to the center of the graphics context.

Listing 9 . Draw a visual manifestation of a GM2D02.Point object.

```
g2Da.translate(osiWidth/2.0,osiHeight/2.0);

GM2D02.Point origin = new GM2D02.Point(
                        new
GM2D02.ColMatrix(0.0,0.0));
origin.draw(g2Da);
```

This is equivalent to changing the coordinate frame described by Kjell and discussed in the previous module.

Then the code in [Listing 9](#) defines a **GM2D02.Point** object located at the origin of the new coordinate frame and calls the **draw** method on that object to produce the visual manifestation of the object in the center of the left image in [Figure 1](#).

Once again, the **GM2D02.Point** object is an underlying data object. The visual manifestation will be produced by an object of the Java standard class named **Ellipse2D.Double**, which will be a completely independent object.

The draw method of the GM2D02.Point class

[Listing 10](#) shows the **draw** method that was added to the **GM2D02.Point** class to produce the updated game-math library. This method draws a small circle around the location of the point on the specified graphics context.

Listing 10 . The draw method of the GM2D02.Point class.

```
public void draw(Graphics2D g2D){
    Ellipse2D.Double circle =
        new
    Ellipse2D.Double(getData(0)-3,
    getData(1)-3,
                                6,
                                6);
    g2D.draw(circle);
} //end draw
```

The logic behind this method is very similar to the logic that I explained relative to [Listing 8](#). The constructor for the Java standard **Ellipse2D.Double** class requires four incoming parameters that specify the coordinates of the upper-left corner of a rectangle followed by the width and the height of the rectangle. The new object of type **Ellipse2D.Double** represents an ellipse that is bounded by the four sides of the rectangle. If the rectangle is square, the ellipse becomes a circle.

(In this case, the rectangle is a 6x6 square, thus producing a circle with a diameter of six pixels.)

[Listing 10](#) calls the **draw** method of the **Graphics2D** class to render the ellipse (*circle*) at the specified location on the graphics context specified by the incoming parameter. Thus the code in [Listing 9](#) produces a visual manifestation of a point at the origin of the current coordinate frame. The visual manifestation consists of a small circle centered on the location of the point, resulting in the small circle at the center of the left image in [Figure 1](#).

Draw the vertices of a hexagon

Returning once more to the **drawOffScreen** method of the program named **PointLine03** , [Listing 11](#) instantiates six **GM2D02.Point** objects that represent the vertices of a hexagon that is symmetrically located relative to the origin in the current coordinate frame

Listing 11 . Draw the vertices of a hexagon.

Listing 11 . Draw the vertices of a hexagon.

```
//First define three constants to make it easier to
// write the code.
final double aVal = osiWidth/4.0*0.5;
final double bVal = osiWidth/4.0*0.866;
final double cVal = osiWidth/4.0;
//Now define the points.
GM2D02.Point point0 = new GM2D02.Point(
    new
GM2D02.ColMatrix(cVal,0.0));
GM2D02.Point point1 = new GM2D02.Point(
    new
GM2D02.ColMatrix(aVal,bVal));
GM2D02.Point point2 = new GM2D02.Point(
    new GM2D02.ColMatrix(-
aVal,bVal));
GM2D02.Point point3 = new GM2D02.Point(
    new GM2D02.ColMatrix(-
cVal,0.0));
GM2D02.Point point4 = new GM2D02.Point(
    new GM2D02.ColMatrix(-aVal,-
bVal));
GM2D02.Point point5 = new GM2D02.Point(
    new GM2D02.ColMatrix(aVal,-
bVal));

//Now draw a visual manifestation of each of the six
// points on g2Da.
point0.draw(g2Da);
point1.draw(g2Da);
point2.draw(g2Da);
point3.draw(g2Da);
point4.draw(g2Da);
point5.draw(g2Da);
```

Then [Listing 11](#) calls the **draw** method of the **GM2D02.Point** class six times in succession to cause small circles that represent the six points to be rendered on the specified off-screen image. You can see those six circles in the left image in [Figure 1](#).

Draw six lines connecting the vertices of the hexagon

[Listing 12](#) instantiates six objects of the **GM2D02.Line** class whose endpoints are specified by the six **GM2D02.Point** objects from [Listing 11](#), taken in pairs.

Listing 12 . Draw six lines connecting the vertices of the hexagon.

```
GM2D02.Line line01 = new GM2D02.Line(point0,point1);
GM2D02.Line line12 = new GM2D02.Line(point1,point2);
GM2D02.Line line23 = new GM2D02.Line(point2,point3);
GM2D02.Line line34 = new GM2D02.Line(point3,point4);
GM2D02.Line line45 = new GM2D02.Line(point4,point5);
GM2D02.Line line50 = new GM2D02.Line(point5,point0);

//Now draw a visual manifestation of each line
// on g2Da.
line01.draw(g2Da);
line12.draw(g2Da);
line23.draw(g2Da);
line34.draw(g2Da);
line45.draw(g2Da);
line50.draw(g2Da);
```

Then [Listing 12](#) calls the **draw** method belonging to the **GM2D02.Line** class six times in succession to cause visible lines to be rendered on the specified off-screen image. You can see those six lines in the left image in [Figure 1](#).

That completes the drawing that is performed on the off-screen image shown in the left image in [Figure 1](#). Next I will explain the drawing that is performed on the off-screen image that is shown as the right image in [Figure 1](#).

Instantiate three objects of type GM2D02.Vector

According to Kjell, "A vector is a geometrical object that has two properties: **length and direction** ." He also tells us, "A vector does not have a position."

The right image in [Figure 1](#) shows visual manifestations of three different objects of the **GM2D02.Vector** class. One vector is represented as a red line with a small red circle at its head. A second vector is represented as a blue line with a small blue circle at its head. The

third vector is shown in three different positions represented by green lines with small green circles at their heads.

[Listing 13](#) instantiates three objects of the **GM2D02.Vector** class, which are visually represented by the red, green, and blue lines in [Figure 1](#). References to these three objects are saved in the variables named **vecA** , **vecB** , and **vecC** in [Listing 13](#).

Listing 13 . Instantiate three objects of type GM2D02.Vector.

```
GM2D02.Vector vecA = new GM2D02.Vector(  
                                new  
GM2D02.ColMatrix(50,100));  
GM2D02.Vector vecB = new GM2D02.Vector(  
                                new  
GM2D02.ColMatrix(75,25));  
GM2D02.Vector vecC = new GM2D02.Vector(  
                                new  
GM2D02.ColMatrix(125,125));
```

Note that each vector is described by two values in a **GM2D02.ColMatrix** object.

Call the draw method to draw the vector

[Listing 14](#) begins by setting the drawing color to red for the off-screen image on which the visual manifestation of the first **Vector** object will be drawn. Then [Listing 14](#) calls the **draw** method of the **GM2D02.Vector** class to cause the object referred to by **vecA** to be represented as a (red) line with its tail at the origin. (Unlike the image on the left side of [Figure 1](#), the origin for the image on the right was not translated to the center.)

Listing 14 . Call the draw method to draw the vector.

Listing 14 . Call the draw method to draw the vector.

```
g2Db.setColor(Color.RED);  
  
vecA.draw(g2Db,new GM2D02.Point(  
                                new  
GM2D02.ColMatrix(0,0)));
```

Note that two parameter are passed to the **draw** method in Listing 14:

- A reference to the off-screen graphics context on which the visual manifestation of the vector will be drawn.
- A new object of the class **GM2D02.Point** that will be used to determine the position on the off-screen image in which the visual manifestation will appear.

Remember that according to Kjell, a vector doesn't have a position. Hence, there is nothing in the underlying data for a **GM2D02.Vector** object that specifies a position. In other words, the visual manifestation of a vector can be placed anywhere in space, and one placement is just as correct as the next. However, if you become heavily involved in the use of vectors, you will learn that certain placements may be preferable to others in some cases so as to better represent the problem being modeled by the use of vectors.

The draw method of the GM2D02.Vector class

Switching once again to the **GM2D02** library, [Listing 15](#) shows the **draw** method that was added to the **GM2D02.Vector** class. This method is a little longer than the **draw** methods that I explained earlier for points and lines. This is mainly because:

- it is necessary to deal with the issue of positioning the visual manifestation of the **GM2D02.Vector** object, and
- it is necessary to embellish the drawing to make it possible to visually determine which end is the tail and which end is the head.

This method renders a visual manifestation of a **GM2D02.Vector** on the specified graphics context, with the tail of the vector located at a point specified by the contents of a **GM2D02.Point** object. A small circle is drawn to visually identify the head of the vector.

Listing 15 . The draw method of the GM2D02.Vector class.

```
public void draw(Graphics2D g2D, GM2D02.Point tail){
    Line2D.Double line = new Line2D.Double(
        tail.getData(0),
        tail.getData(1),
        tail.getData(0)+vector.getData(0),
        tail.getData(1)+vector.getData(1));

    //Draw a small circle to identify the head.
    Ellipse2D.Double circle = new Ellipse2D.Double(
        tail.getData(0)+vector.getData(0)-2,
        tail.getData(1)+vector.getData(1)-2,
        4,
        4);
    g2D.draw(circle);
    g2D.draw(line);
} //end draw
```

Why a circle and not an arrowhead?

When we draw vectors, (*particularly when drawing by hand*) , we often draw a small arrowhead at the head of the vector. Remember, a vector has only two properties: *length* and [direction](#). The arrowhead normally points in the direction indicated by the *direction* property. However, there is nothing magic about an arrowhead. Circles are much easier to draw with a computer than arrowheads. As long as you can identify the head of the vector, you can always determine the direction. Although I didn't give a lot of thought to optimization when developing this library, this is one case where I took the approach that is likely to consume the minimum amount of computational resources.

In [Figure 1](#), the length of each of the lines in the right image indicates the *length* property of the respective vector, and the small circle indicates the head. The orientation relative to the current coordinate frame represents the *direction* .

Beyond that, further explanation of the code in [Listing 15](#) should not be required.

Three visual manifestations of the same vector

Returning again to the **drawOffScreen** method of the program named **PointLine03** , [Listing 16](#) produces three visual manifestations at three different positions for the same

Vector object referred to by the reference variable named **vecB** . All three of the visual manifestations are colored green in [Figure 1](#).

Listing 16 . Three visual manifestations of the same vector.

```
g2Db.setColor(Color.GREEN);
vecB.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
vecA.getData(0),vecA.getData(1))));
vecB.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
vecA.getData(0)-10,vecA.getData(1)+15)));
vecB.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
vecA.getData(0)+10,vecA.getData(1)-60)));
```

This is a perfectly legitimate thing to do since the underlying data encapsulated in the **Vector** object does not contain any information relating to the position of the visual manifestation.

In one visual manifestation, [Listing 16](#) causes the tail of **vecB** to coincide with the head of **vecA** . This is one of those cases where drawing the vector in one position is preferable to drawing it in a different position. Later on when we get into the topic of vector addition, I will explain that one way to perform graphical addition of vectors is to position the vectors in a tail-to-head relationship as indicated by the red and green vectors in [Figure 1](#). The sum of the vectors can then be determined graphically by drawing a line from the tail of the first vector to the head of the last vector as indicated by the blue line in [Figure 1](#). The length of the sum (*or resultant*) vector is indicated by the length of that line, and the direction of the resultant vector is indicated by the orientation of that line.

A practical example

This is the solution to a classical problem in a freshman engineering class. Assume that a boat is traveling diagonally across a river with a strong current. Assume that the green vector in [Figure 1](#) represents the speed and direction of the current in the river. Assume that the red vector represents the speed and direction that the boat would travel if there were no current. Because of the effect of the current on the boat, the actual speed and direction of the boat will be given by the blue vector, which is different from the speed and direction

indicated by the red vector. If the blue vector is pointing at the desired landing point on the other side of the river, everything is okay. Otherwise, the captain needs to change the speed and/or the direction of the boat to compensate for the effect of the current on the boat.

Draw the blue vector

[Listing 17](#) produces the blue visual manifestation of the **Vector** object referred to by **vecC** as shown in [Figure 1](#).

Listing 17 . Draw the blue vector.

```
g2Db.setColor(Color.BLUE);
vecC.draw(g2Db,new GM2D02.Point(
                                new
GM2D02.ColMatrix(0,0)));
} //end drawOffscreen
```

[Listing 17](#) also signals the end of the method named **drawOffscreen** .

The MyCanvas class and the overridden paint method

[Listing 18](#) shows the entire inner class named **MyCanvas** including the overridden **paint** method belonging to that class.

Listing 18 . The MyCanvas class and the overridden paint method.

Listing 18 . The MyCanvas class and the overridden paint method.

```
//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    public void paint(Graphics g){
        g.drawImage(osiA,0,0,this);
        g.drawImage(osiB,this.getWidth()/2,0,this);
    }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
```

The overridden **paint** method will be called when the **JFrame** and the **Canvas** appear on the screen or when the **repaint** method is called on the **Canvas** object.

The method is also called when some screen activity requires the **JFrame** to be redrawn (such as minimizing and then restoring the **JFrame**).

The purpose of the **paint** method in this program is to call the **drawImage** method twice in succession to draw the two off-screen images on the screen in the side-by-side format shown in [Figure 1](#).

The first parameter to the **drawImage** method is the off-screen image that is to be drawn on the canvas. The second and third parameters specify the location where the upper-left corner of the image will be drawn.

The fourth parameter is a reference to an **ImageObserver** object, which is essentially a dummy image observer in this case, because an actual image observer isn't needed. (To learn more about image observers, see [The AWT Package, Graphics- Introduction to Images](#).)

End of the program

[Listing 18](#) also signals the end of the GUI class and the end of the program named **PointLine03** ..

The program named PointLine04

A complete listing of this program is provided in [Listing 28](#). Just like the previous program named **PointLine03** , this program produces the screen output shown in [Figure 1](#).

However, it produces that output in a significantly different way.

This program emphasizes the differences between graphics objects and underlying data objects. It also illustrates the use of the new **setData** methods of the **Point** class and the **Vector** class, along with the new **setTail** and **setHead** methods of the **Line** class. These methods were added to the game-math library when it was updated to the **GM2D02** version.

Dealing with some vulnerabilities

The addition of the new *set* methods exposed some vulnerabilities in the original version of the game-math library. I will explain how the code in the library was modified to deal with that issue. In particular, constructors were updated to ensure that existing points, vectors, and lines are not corrupted by using the new *set* methods to change the values in the **ColMatrix** and/or **Point** objects originally used to construct the points, vectors, and lines. The updated constructors create and save clones of the **ColMatrix** and/or **Point** objects originally used to define the **Point** , **Vector** , and/or **Line** objects.

Will explain the code in fragments

As before, I will explain the code in fragments. Some of the code in this program is identical to or very similar to code that I have already explained in the program named **PointLine03** . I won't repeat that explanation. Instead, I will concentrate on the differences between the two programs.

The setData method of the ColMatrix class

The new **setData** method of the **ColMatrix** class is shown in [Listing 19](#). Note that this code is part of the **GM2D02** library. At this point, I will begin switching back and forth between the library and the program named **PointLine04** .

Listing 19 . The setData method of the ColMatrix class.

Listing 19 . The setData method of the ColMatrix class.

```
public void setData(int index,double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        this.data[index] = data;
    }//end else
}//end setData method
```

This is a very simple method, which should not require further explanation. I am presenting it here mainly for the purpose of leading into a discussion of the vulnerabilities that this method could expose on **Point** and **Vector** objects instantiated from the original math class named **GM2D01** ,

You may recall from the module titled *Getting Started* that the constructor for the **Point** class in the original library received and saved a reference to an object of the **ColMatrix** class that contained the two values required to define the **Point** . Now that it is possible to change the values stored in an object of the **ColMatrix** class, if such an object is used to instantiate an object of the **Point** class and then the values stored in the **ColMatrix** object are changed, the values that define the **Point** object will change accordingly. That is not good.

Updated constructor for the Point class in GM2D02

Continuing with code in the **GM2D02** library, [Listing 20](#) shows the updated constructor in the new version of the library.

Listing 20 . Updated constructor for the Point class in GM2D02.

Listing 20 . Updated constructor for the Point class in GM2D02.

```
public static class Point{
    GM2D02.ColMatrix point;

    Point(GM2D02.ColMatrix point){//constructor
        this.point =
            new
            ColMatrix(point.getData(0),point.getData(1));
    }//end constructor
```

The new version of the constructor for the **Point** class creates and saves a clone of the **ColMatrix** object used to define the point to prevent the point from being corrupted by a later change in the values stored in the original **ColMatrix** object through the use of its new **setData** method.

Updated constructor for the Vector class in GM2D02

Because the constructor for the **Vector** class in the original library was essentially the same as the constructor for the **Point** class, the addition of the **setData** method to the **ColMatrix** class created the same vulnerability for objects instantiated from the **Vector** class. Essentially the same correction was made for the constructor for the **Vector** class. You can view the new code in [Listing 26](#).

Additional changes to the library

In addition to the changes described above, the following changes were made to the game-math library to update it from version **GM2D01** to **GM2D02** . All of the changes were straightforward and shouldn't require an explanation beyond the embedded comments. You can view the new code for all of these changes in [Listing 26](#).

- Added a **setData** method for the **Point** class.
- Updated the constructor for the **Line** class to deal with the same kind of vulnerability described above.
- Added a **setTail** method for the **Line** class.
- Added a **setHead** method for the **Line** class.

The drawOffscreen method of the program named PointLine04

Most of the differences between the programs named **PointLine03** and **PointLine04** occur in the method named **drawOffscreen** . I will concentrate my discussion there.

Switching now to code in the program named **PointLine04** , the **drawOffscreen** method begins in [Listing 21](#).

Listing 21 . The drawOffscreen method of the program named PointLine04.

```
void drawOffscreen(Graphics2D g2Da,Graphics2D g2Db){  
    //Draw a label on each off-screen image.  
    g2Da.drawString("Off-screen image A",  
                    osiWidth/8,  
                    osiHeight/8);  
    g2Db.drawString("Off-screen image B",  
                    osiWidth/8,  
                    osiHeight/8);  
}
```

As before, the purpose of the **drawOffscreen** method is to define points, lines, and vectors and then to cause a visual manifestation of some of the points, lines, and vectors to be drawn onto two separate off-screen images. Unlike before, however, this version of the method uses a minimum number of underlying data objects to produce the output, thereby emphasizing the differences between graphics objects and underlying data objects.

The code in [Listing 21](#) is essentially the same as before and doesn't require any explanation.

Borders

As before, and as shown in [Figure 1](#), this method draws borders on each of the off-screen images by drawing lines parallel to the edges of the off-screen images. Each line is offset by one pixel toward the center of the off-screen image.

Instantiate a point at the upper-left corner

[Listing 22](#) instantiates and saves an underlying data object of type **Point** containing coordinate values for the upper left corner. This object will be used as one endpoint for a *moveable* **Line** object from which the top and left borders will be drawn.

Listing 22 . Instantiate a point at the upper-left corner.

```
GM2D02.Point upperLeftPoint = new GM2D02.Point(  
                                new  
GM2D02.ColMatrix(1.0,1.0));
```

Instantiate a moveable Point object

[Listing 23](#) instantiates a **Point** object that will be used as an endpoint for a moveable line in all four locations.

Listing 23 . Instantiate a moveable Point object.

```
GM2D02.Point aPoint = new GM2D02.Point(  
    new GM2D02.ColMatrix(osiWidth-1,  
upperLeftPoint.getData(1)));
```

The location of this point will be modified several times during the drawing of the border by calling the **setData** method on the point. The initial location of this point is the upper-right corner. *(Recall that the origin is at the upper-left corner of both off-screen images at this point in the program.)* The call to the **Point** constructor causes the y-coordinate to be the same as the y-coordinate for the **Point** object instantiated in [Listing 22](#).

Instantiate a moveable Line object

[Listing 24](#) instantiates a **Line** object that will be modified several times in succession by calling its **setHead** and **setTail** methods to place it in four different locations.

Listing 24 . Instantiate a moveable Line object.

Listing 24 . Instantiate a moveable Line object.

```
GM2D02.Line theLine = new GM2D02.Line(  
upperLeftPoint, aPoint);  
theLine.draw(g2Da);  
theLine.draw(g2Db);
```

This **Line** object will be used to draw the top, right, bottom, and left lines that make up the border. It is used in [Listing 24](#) to cause a graphical **Line2D.Double** object to be instantiated and rendered as the line at the top of both of the off-screen images.

Relocate the Line to three more locations

[Listing 25](#) calls various methods of the **Point** and **Line** classes (*including **setData** , **setTail** , and **setHead***) to relocate the **GM2D02.Line** object to three more locations, each parallel to an edge of an off-screen image.

Listing 25 . Relocate the Line to three more locations.

Listing 25 . Relocate the Line to three more locations.

```
//Draw right border.
//Save the previous head as the new tail.
theLine.setTail(theLine.getHead());
//Modify the location of aPoint. There's no need to
// change the x-coordinate. Just change the
// y-coordinate to move the point down the screen in
// the positive y direction to the lower-right
corner.
aPoint.setData(1,osiHeight-1);
//Use the new location of aPoint as the new head.
theLine.setHead(aPoint);
theLine.draw(g2Da);
theLine.draw(g2Db);

//Draw bottom border. There's no need to change the
// y-coordinate in the new point, which is located
// at the lower-left corner.
theLine.setTail(theLine.getHead());
aPoint.setData(0,1.0);
theLine.setHead(aPoint);
theLine.draw(g2Da);
theLine.draw(g2Db);

//Draw left border to close the rectangular border.
theLine.setTail(theLine.getHead());
theLine.setHead(upperLeftPoint);
theLine.draw(g2Da);
theLine.draw(g2Db);
```

Draw the border

While the single **GM2D02.Line** object is at each location, **draw** methods are called to cause a **Line2D.Double** object to be instantiated and rendered on each of the off-screen images. This results in four different visual manifestations of the single **GM2D02.Line** object, producing visible lines at the edges of each off-screen image. Although this code is not particularly straightforward, the embedded comments should suffice to explain it.

Draw the hexagon

Essentially the same process is used to draw the hexagon shown in the left image in [Figure 1](#). You can view the code that does that in [Listing 28](#). Once you understand the code that I explained above, you should have no difficulty understanding the code that draws the hexagon.

Draw the vectors

A very similar process is used to draw the vectors shown in the right image in [Figure 1](#). You can view the code in [Listing 28](#). Once again, if you understand the code that I explained above, you should have no difficulty understanding the code that draws the vectors.

That ends the explanation of the program named **PointLine04**.

Documentation for the GM2D02 library

Click [here](#) to download a zip file containing standard javadoc documentation for the library named **GM2D02**. Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text (see [Listing 26](#) and the explanations given above), the documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

Homework assignment

Your homework assignment for this module was to study Kjell's *CHAPTER 0 -- Points and Lines* plus *CHAPTER 1 -- Vectors, Points, and Column Matrices* down through the topic titled *Variables as Elements*.

The homework assignment for the next module is to study the Kjell tutorial through *Chapter 2 -- Column and Row Matrix Addition*.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 26](#), [Listing 27](#), and [Listing 28](#), compile it, and execute it. Experiment with it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to update the game-math library to provide a number of new capabilities including the addition of graphics to the library and the addition of **set** methods for column matrices, points, vectors, and lines.

You also saw sample programs that illustrate the use of those new capabilities and you learned how to draw on off-screen images.

What's next?

In the next module, you will learn how to compare column matrices for equality, how to compare two points for equality, how to compare two vectors for equality, how to add one column matrix to another, how to subtract one column matrix from another, and how to get a displacement vector from one point to another.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME2302-0110: Updating the Game Math Library for Graphics
- File: Game0110.htm
- Published: 10/14/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 26](#) through [Listing 28](#) below.

Listing 26 . Source code for the game-math library class named GM2D02.

```
/*GM2D02.java  
Copyright 2008, R.G.Baldwin  
Revised 02/03/08
```

The name GM2Dnn is an abbreviation for GameMath1Dnn.

See the file named GM2D01.java for a general description of this game-math library file. This file is an update of GM2D01.

This update added the following new capabilities:

- Draw circles around points
- Draw lines
- Draw vectors
- Call a set method to change the values in a ColMatrix object.
- Call a set method to change the values in a Point object.
- Call a set method to change the values in a Vector object.
- Call a set method to change one of the points that defines a line.

Constructors were updated to ensure that existing points, vectors, and lines are not corrupted by using the new set methods to change the values in the ColMatrix and/or

Point objects originally used to construct the points, vectors, and lines.

The updated constructors create and save clones of the ColMatrix and/or Point objects originally used to define the Point, Vector, and/or Line objects.

Tested using JDK 1.6 under WinXP.

```
*****/
```

```
import java.awt.geom.*;
```

```
import java.awt.*;
```

```
public class GM2D02{
```

```
    //An object of this class represents a 2D column matrix.
    // An object of this class is the fundamental building
    // block for several of the other classes in the
    // library.
```

```
    public static class ColMatrix{
        double[] data = new double[2];
```

```
        ColMatrix(double data0,double data1){//constructor
            data[0] = data0;
            data[1] = data1;
        }//end constructor
```

```
        //-----//
```

```
        public String toString(){
            return data[0] + "," + data[1];
        }//end overridden toString method
```

```
        //-----//
```

```
        public double getData(int index){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                return data[index];
            }//end else
```

```
        }//end getData method
```

```
        //-----//
```

```
        public void setData(int index,double data){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
```



```

        this.data[index] = data;
    }//end else
} //end setData method
//-----//
} //end class ColMatrix
//=====//

public static class Point{
    GM2D02.ColMatrix point;

    Point(GM2D02.ColMatrix point){//constructor
        //Create and save a clone of the ColMatrix object
        // used to define the point to prevent the point
        // from being corrupted by a later change in the
        // values stored in the original ColMatrix object
        // through use of its set method.
        this.point =
            new ColMatrix(point.getData(0),point.getData(1));
    } //end constructor
    //-----//

    public String toString(){
        return point.getData(0) + "," + point.getData(1);
    } //end toString
    //-----//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        } else{
            return point.getData(index);
        } //end else
    } //end getData
    //-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        } else{
            point.setData(index,data);
        } //end else
    } //end setData
    //-----//

    //This method draws a small circle around the location

```

```

// of the point on the specified graphics context.
public void draw(Graphics2D g2D){
    Ellipse2D.Double circle =
        new Ellipse2D.Double(getData(0)-3,
                             getData(1)-3,
                             6,
                             6);

    g2D.draw(circle);
} //end draw
//-----//
} //end class Point
//=====//

public static class Vector{
    GM2D02.ColMatrix vector;

    Vector(GM2D02.ColMatrix vector){ //constructor
        //Create and save a clone of the ColMatrix object
        // used to define the vector to prevent the vector
        // from being corrupted by a later change in the
        // values stored in the original ColMatrix object.
        this.vector = new ColMatrix(
            vector.getData(0), vector.getData(1));
    } //end constructor
    //-----//

    public String toString(){
        return vector.getData(0) + "," + vector.getData(1);
    } //end toString
    //-----//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        } else{
            return vector.getData(index);
        } //end else
    } //end getData
    //-----//

    public void setData(int index, double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        } else{
            vector.setData(index, data);
        }
    }
}

```

```

        }//end else
    }//end setData
    //-----//

    //This method draws a vector on the specified graphics
    // context, with the tail of the vector located at a
    // specified point, and with a small circle at the
    // head.
    public void draw(Graphics2D g2D,GM2D02.Point tail){
        Line2D.Double line = new Line2D.Double(
            tail.getData(0),
            tail.getData(1),
            tail.getData(0)+vector.getData(0),
            tail.getData(1)+vector.getData(1));

        //Draw a small circle to identify the head.
        Ellipse2D.Double circle = new Ellipse2D.Double(
            tail.getData(0)+vector.getData(0)-2,
            tail.getData(1)+vector.getData(1)-2,
            4,
            4);

        g2D.draw(circle);
        g2D.draw(line);
    }//end draw
    //-----//
}//end class Vector
//=====//

//A line is defined by two points. One is called the
// tail and the other is called the head.
public static class Line{
    GM2D02.Point[] line = new GM2D02.Point[2];

    Line(GM2D02.Point tail,GM2D02.Point head){
        //Create and save clones of the points used to
        // define the line to prevent the line from being
        // corrupted by a later change in the coordinate
        // values of the points.
        this.line[0] = new Point(new GM2D02.ColMatrix(
            tail.getData(0),tail.getData(1)));
        this.line[1] = new Point(new GM2D02.ColMatrix(
            head.getData(0),head.getData(1)));
    }//end constructor
    //-----//

```

```

public String toString(){
    return "Tail = " + line[0].getData(0) + ", "
        + line[0].getData(1) + "\nHead = "
        + line[1].getData(0) + ", "
        + line[1].getData(1);
} //end toString
//-----//

public GM2D02.Point getTail(){
    return line[0];
} //end getTail
//-----//

public GM2D02.Point getHead(){
    return line[1];
} //end getHead
//-----//

public void setTail(GM2D02.Point newPoint){
    //Create and save a clone of the new point to
    // prevent the line from being corrupted by a
    // later change in the coordinate values of the
    // point.
    this.line[0] = new Point(new GM2D02.ColMatrix(
        newPoint.getData(0), newPoint.getData(1)));
} //end setTail
//-----//

public void setHead(GM2D02.Point newPoint){
    //Create and save a clone of the new point to
    // prevent the line from being corrupted by a
    // later change in the coordinate values of the
    // point.
    this.line[1] = new Point(new GM2D02.ColMatrix(
        newPoint.getData(0), newPoint.getData(1)));
} //end setHead
//-----//

public void draw(Graphics2D g2D){
    Line2D.Double line = new Line2D.Double(
        getTail().getData(0),
        getTail().getData(1),
        getHead().getData(0),
        getHead().getData(1));

    g2D.draw(line);
}

```

```

        }//end draw
        //-----//
    }//end class Line
    //=====//

}//end class GM2D02

```

Listing 27 . Source code for the program named PointLine03.

```

/*PointLine03.java
Copyright 2008, R.G.Baldwin
Revised 02/03/08

```

This program illustrates the use of draw methods that were added to the GM2D02 game-math library to produce visual manifestations of point, line, and vector objects instantiated from classes in the game-math library named GM2D02.

The program also illustrates drawing on off-screen images and then copying those images to a Canvas object in an overridden paint method.

Tested using JDK 1.6 under WinXP.

```

*****/
import java.awt.*;
import javax.swing.*;

class PointLine03{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class PointLine03
//=====//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a JFrame
    // object.
    int hSize = 400;
    int vSize = 200;
    Image osiA;//one off-screen image
    Image osiB;//another off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

```

```

GUI(){//constructor
    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create a new drawing canvas and add it to the
    // center of the JFrame.
    myCanvas = new MyCanvas();
    this.getContentPane().add(myCanvas);

    //This object must be visible before you can get an
    // off-screen image. It must also be visible before
    // you can compute the size of the canvas.
    setVisible(true);
    osiWidth = myCanvas.getWidth()/2;
    osiHeight = myCanvas.getHeight();

    //Create two off-screen images and get a graphics
    // context on each.
    osiA = createImage(osiWidth,osiHeight);
    Graphics2D g2Da = (Graphics2D)(osiA.getGraphics());

    osiB = createImage(osiWidth,osiHeight);
    Graphics2D g2Db = (Graphics2D)(osiB.getGraphics());

    //Draw some points, lines, and vectors on the two
    // off-screen images.
    drawOffscreen(g2Da,g2Db);

    //Cause the overridden paint method belonging to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
//-----//

//The purpose of this method is to define points, lines,
// and vectors and then to cause a visual manifestation
// of some of the points, lines, and vectors to be
// drawn onto two separate off-screen images.
void drawOffscreen(Graphics2D g2Da,Graphics2D g2Db){

    //Draw a label on each off-screen image.

```

```

g2Da.drawString("Off-screen image A",
                osiWidth/8,
                osiHeight/8);
g2Db.drawString("Off-screen image B",
                osiWidth/8,
                osiHeight/8);

//Define four lines that can be used to draw borders
// on each of the off-screen images.
//First define four points that will be used to define
// the ends of the four lines.
GM2D02.Point upperLeftPoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(1.0,1.0));
GM2D02.Point upperRightPoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(osiWidth-1,1.0));
GM2D02.Point lowerRightPoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(osiWidth-1,osiHeight-1));
GM2D02.Point lowerLeftPoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(1.0,osiHeight-1));

//Now define the four lines based on the endpoints..
GM2D02.Line top = new GM2D02.Line(upperLeftPoint,
                                   upperRightPoint);
GM2D02.Line rightSide = new GM2D02.Line(
                                   upperRightPoint,
                                   lowerRightPoint);
GM2D02.Line bottom = new GM2D02.Line(lowerLeftPoint,
                                       lowerRightPoint);
GM2D02.Line leftSide = new GM2D02.Line(upperLeftPoint,
                                       lowerLeftPoint);

//Now draw a visual manifestation of each line
// on g2Da.
top.draw(g2Da);
rightSide.draw(g2Da);
bottom.draw(g2Da);
leftSide.draw(g2Da);

//Now draw a visual manifestation of each of the same
// four lines on g2Db
top.draw(g2Db);
rightSide.draw(g2Db);
bottom.draw(g2Db);
leftSide.draw(g2Db);

```

```

//Translate the origin of g2Da to the center of the
// off-screen image.
g2Da.translate(osiWidth/2.0,osiHeight/2.0);

//Define a point at the new origin and draw a visual
// manifestation of the point.
GM2D02.Point origin = new GM2D02.Point(
                                new GM2D02.ColMatrix(0.0,0.0));
origin.draw(g2Da);

//Define six points that define the vertices of a
// hexagon that is symmetrically located relative to
// the origin. Begin at the right and move clockwise
// around the origin.
//First define three constants to make it easier to
// write the code.
final double aVal = osiWidth/4.0*0.5;
final double bVal = osiWidth/4.0*0.866;
final double cVal = osiWidth/4.0;
//Now define the points.
GM2D02.Point point0 = new GM2D02.Point(
                                new GM2D02.ColMatrix(cVal,0.0));
GM2D02.Point point1 = new GM2D02.Point(
                                new GM2D02.ColMatrix(aVal,bVal));
GM2D02.Point point2 = new GM2D02.Point(
                                new GM2D02.ColMatrix(-aVal,bVal));
GM2D02.Point point3 = new GM2D02.Point(
                                new GM2D02.ColMatrix(-cVal,0.0));
GM2D02.Point point4 = new GM2D02.Point(
                                new GM2D02.ColMatrix(-aVal,-bVal));
GM2D02.Point point5 = new GM2D02.Point(
                                new GM2D02.ColMatrix(aVal,-bVal));

//Now draw a visual manifestation of each of the six
// points on g2Da.
point0.draw(g2Da);
point1.draw(g2Da);
point2.draw(g2Da);
point3.draw(g2Da);
point4.draw(g2Da);
point5.draw(g2Da);

//Now define six lines using the six points taken in
// pairs to define the endpoints of the lines.
GM2D02.Line line01 = new GM2D02.Line(point0,point1);

```



```

GM2D02.Line line12 = new GM2D02.Line(point1,point2);
GM2D02.Line line23 = new GM2D02.Line(point2,point3);
GM2D02.Line line34 = new GM2D02.Line(point3,point4);
GM2D02.Line line45 = new GM2D02.Line(point4,point5);
GM2D02.Line line50 = new GM2D02.Line(point5,point0);

//Now draw a visual manifestation of each line
// on g2Da.
line01.draw(g2Da);
line12.draw(g2Da);
line23.draw(g2Da);
line34.draw(g2Da);
line45.draw(g2Da);
line50.draw(g2Da);

//Now define three vectors and draw visual
// manifestations of the vectors on g2Db.
GM2D02.Vector vecA = new GM2D02.Vector(
                                new GM2D02.ColMatrix(50,100));
GM2D02.Vector vecB = new GM2D02.Vector(
                                new GM2D02.ColMatrix(75,25));
GM2D02.Vector vecC = new GM2D02.Vector(
                                new GM2D02.ColMatrix(125,125));

//Draw vecA in red with its tail at the origin, which
// is still at the upper-left corner of the g2Db
// off-screen image
g2Db.setColor(Color.RED);
vecA.draw(g2Db,new GM2D02.Point(
                                new GM2D02.ColMatrix(0,0)));

//Draw three visual manifestations of vecB. Cause
// the tail of vecB to coincide with the head of vecA
// in one of the three visual manifestations. Make all
// three of them green.
g2Db.setColor(Color.GREEN);
vecB.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
                                vecA.getData(0),vecA.getData(1))));
vecB.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
                                vecA.getData(0)-10,vecA.getData(1)+15)));
vecB.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
                                vecA.getData(0)+10,vecA.getData(1)-60)));

//Draw vecC in blue with its tail at the origin.

```

```

        g2Db.setColor(Color.BLUE);
        vecC.draw(g2Db,new GM2D02.Point(
                                new GM2D02.ColMatrix(0,0)));

    }//end drawOffscreen
    //=====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method will be
        // called when the JFrame and the Canvas appear on the
        // screen or when the repaint method is called on the
        // Canvas object.
        //The purpose of this method is to display the two
        // off-screen images on the screen in a side-by-side
        // format.
        public void paint(Graphics g){
            g.drawImage(osiA,0,0,this);
            g.drawImage(osiB,this.getWidth()/2,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI

```

Listing 28 . Source code for the program named PointLine04.

```

/*PointLine04.java
Copyright 2008, R.G.Baldwin
Revised 02/03/08

```

This program emphasizes the differences between graphics objects and underlying data objects. It also illustrates the use of the new setData methods of the Point class and the Vector class, along with the new setTail and setHead methods of the Line class. The methods were added to the game-math library when it was updated to the GM2D02 version.

This program produces the same graphic output as that produced by PointLine03, but it produces that output in a significantly different way.

Tested using JDK 1.6 under WinXP.

```

*****/
import java.awt.*;
import javax.swing.*;

class PointLine04{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class PointLine04
//=====================================================//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a JFrame
    // object.
    int hSize = 400;
    int vSize = 200;
    Image osiA;//one off-screen image
    Image osiB;//another off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        //This object must be visible before you can get an
        // off-screen image. It must also be visible before
        // you can compute the size of the canvas.
        setVisible(true);
        osiWidth = myCanvas.getWidth()/2;
        osiHeight = myCanvas.getHeight();

        //Create two off-screen images and get a graphics
        // context on each.
        osiA = createImage(osiWidth,osiHeight);
        Graphics2D g2Da = (Graphics2D)(osiA.getGraphics());

```

```

osiB = createImage(osiWidth,osiHeight);
Graphics2D g2Db = (Graphics2D)(osiB.getGraphics());

//Draw some points, lines, and vectors on the two
// off-screen images.
drawOffscreen(g2Da,g2Db);

//Cause the overridden paint method belonging to
// myCanvas to be executed.
myCanvas.repaint();

}//end constructor
//-----//

//The purpose of this method is to define points, lines,
// and vectors and then to cause a visual manifestation
// of some of the points, lines, and vectors to be
// drawn onto two separate off-screen images. The method
// uses a minimum number of underlying data objects to
// produce the output, thereby emphasizing the
// differences between graphics objects and underlying
// data objects.
void drawOffscreen(Graphics2D g2Da,Graphics2D g2Db){

    //Draw a label on each off-screen image.
    g2Da.drawString("Off-screen image A",
                    osiWidth/8,
                    osiHeight/8);
    g2Db.drawString("Off-screen image B",
                    osiWidth/8,
                    osiHeight/8);

    //Draw borders on each of the off-screen images by
    // drawing lines parallel to the edges of the
    // off-screen. Each line is offset by one pixel toward
    // the center of the off-screen image.
    //First define and save a point at the upper left
    // corner that will be used as an endpoint of the
    // top and left borders.
    GM2D02.Point upperLeftPoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(1.0,1.0));

    //Define a point that will be used as an endpoint for
    // all four lines. The location of this point will be
    // modified several times during the drawing of the

```

```

// border by calling the set method on the point. The
// initial location of this point is the upper-right
// corner. Remember, the origin is at the upper-left
// corner of the off-screen images at this point in
// the program. Make the y-coordinate the same as the
// y-coordinate for the upperLeftPoint.
GM2D02.Point aPoint = new GM2D02.Point(
    new GM2D02.ColMatrix(osiWidth-1,
        upperLeftPoint.getData(1)));

//Define a line that will be modified several times in
// succession by calling its set methods to draw the
// top, right, bottom, and left lines that make up the
// border. Use it here to draw the line at the top
// of both of the off-screen images.
GM2D02.Line theLine = new GM2D02.Line(
    upperLeftPoint,aPoint);

theLine.draw(g2Da);
theLine.draw(g2Db);

//Draw right border.
//Save the previous head as the new tail.
theLine.setTail(theLine.getHead());
//Modify the location of aPoint. There's no need to
// change the x-coordinate. Just change the
// y-coordinate to move the point down the screen in
// the positive y direction to the lower-right corner.
aPoint.setData(1,osiHeight-1);
//Use the new location of aPoint as the new head.
theLine.setHead(aPoint);
theLine.draw(g2Da);
theLine.draw(g2Db);

//Draw bottom border. There's no need to change the
// y-coordinate in the new point, which is located
// at the lower-left corner.
theLine.setTail(theLine.getHead());
aPoint.setData(0,1.0);
theLine.setHead(aPoint);
theLine.draw(g2Da);
theLine.draw(g2Db);

//Draw left border to close the rectangular border.
theLine.setTail(theLine.getHead());
theLine.setHead(upperLeftPoint);

```

```

theLine.draw(g2Da);
theLine.draw(g2Db);

//Translate the origin of g2Da to the center of the
// off-screen image.
g2Da.translate(osiWidth/2.0,osiHeight/2.0);

//Define a point at the new origin and draw a visual
// manifestation of the point.
GM2D02.Point origin = new GM2D02.Point(
                                new GM2D02.ColMatrix(0.0,0.0));
origin.draw(g2Da);

//Define two points and one line and use them to
// define and draw a hexagon that is symmetrically
// located relative to the origin.  Begin at the right
// and move clockwise around the origin.

//First establish three constants to make it easier
// to write the code.
final double aVal = osiWidth/4.0*0.5;
final double bVal = osiWidth/4.0*0.866;
final double cVal = osiWidth/4.0;

//The coordinates of the following point are modified
// several times in succession to define the vertices
// of the hexagon.  Instantiate the point and draw it.
GM2D02.Point thePoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(cVal,0.0));
thePoint.draw(g2Da);

//Save a clone of thePoint to be used to draw the
// first line and the last line that closes the
// hexagon.
GM2D02.Point startAndEndPoint = new GM2D02.Point(
                                new GM2D02.ColMatrix(
                                    thePoint.getData(0),thePoint.getData(1)));

//Now call the set method twice to modify the location
// of thePoint, Draw thePoint, and use it to
// instantiate and draw a line from the starting
// location to the new location.
thePoint.setData(0,aVal);
thePoint.setData(1,bVal);

```

```

thePoint.draw(g2Da);
GM2D02.Line aLine = new GM2D02.Line(
                                startAndEndPoint,thePoint);
aLine.draw(g2Da);

//Repeat the process four more times using
// startAndEndPoint, thePoint, and aLine to draw the
// remaining vertices and lines.
//Modify aLine, saving the previous head as the new
// tail.
aLine.setTail(aLine.getHead());
//Modify the location of aPoint and draw it. There's
// no need to change the y-coordinate of the point.
thePoint.setData(0,-aVal);
thePoint.draw(g2Da);
//Modify the endpoint of aLine and draw it.
aLine.setHead(thePoint);
aLine.draw(g2Da);

aLine.setTail(aLine.getHead());
thePoint.setData(0,-cVal);
thePoint.setData(1,0.0);
thePoint.draw(g2Da);
aLine.setHead(thePoint);
aLine.draw(g2Da);

aLine.setTail(aLine.getHead());
thePoint.setData(0,-aVal);
thePoint.setData(1,-bVal);
thePoint.draw(g2Da);
aLine.setHead(thePoint);
aLine.draw(g2Da);

aLine.setTail(aLine.getHead());
thePoint.setData(0,aVal);
//No need to change the y-coordinate.
thePoint.draw(g2Da);
aLine.setHead(thePoint);
aLine.draw(g2Da);

//Now modify and draw aLine to close the hexagon.
aLine.setTail(aLine.getHead());
aLine.setHead(startAndEndPoint);
aLine.draw(g2Da);

```

```

//Now define a vector, call the set method to modify
// it several times, and draw different visual
// manifestations of the vector on g2Db.
//First define two constants to make it easier to
// write the code.
final double firstTail = 50;
final double firstHead = 100;

GM2D02.Vector vec = new GM2D02.Vector(
    new GM2D02.ColMatrix(firstTail,firstHead));
//Draw vec in red with its tail at the origin, which
// is still at the upper-left corner of the g2Db
// off-screen image
g2Db.setColor(Color.RED);
vec.draw(g2Db,new GM2D02.Point(
    new GM2D02.ColMatrix(0,0)));

//Call the set method to modify vec.
vec.setData(0,75);
vec.setData(1,25);
//Draw three visual manifestations of vec. Cause
// the tail to coincide with the head of the previous
// visual manifestation of vec in one of the three new
// visual manifestations. Make all three of them
// green.
g2Db.setColor(Color.GREEN);
vec.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
    firstTail,firstHead)));
vec.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
    firstTail-10,firstHead+15)));
vec.draw(g2Db,new GM2D02.Point(new GM2D02.ColMatrix(
    firstTail+10,firstHead-60)));

//Call the set method to modify vec again.
vec.setData(0,125);
vec.setData(1,125);

//Draw the modified vec in blue with its tail at the
// origin.
g2Db.setColor(Color.BLUE);
vec.draw(g2Db,new GM2D02.Point(
    new GM2D02.ColMatrix(0,0)));
} //end drawOffScreen

```



```
//=====//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method will be
    // called when the JFrame and the Canvas appear on the
    // screen or when the repaint method is called on the
    // Canvas object.
    //The purpose of this method is to display the two
    // off-screen images on the screen in a side-by-side
    // format.
    public void paint(Graphics g){
        g.drawImage(osiA,0,0,this);
        g.drawImage(osiB,this.getWidth()/2,0,this);
    }//end overridden paint()

} //end inner class MyCanvas

} //end class GUI
```

Exercises

Exercise 1

Using Java and the game-math library named **GM2D02** , or using a different programming environment of your choice, write a program that draws a horizontal line segment and a vertical line segment representing the x and y axes in a 2D Cartesian coordinate system as shown in [Figure 2](#).

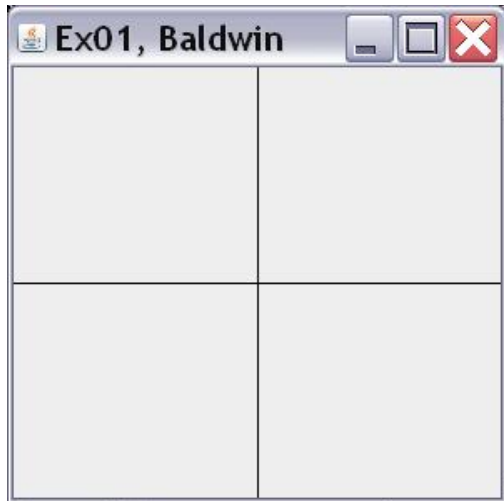
Cause the origin of your reference frame be at the center of your drawing.

Cause the positive x direction be to the right.

Cause the positive y direction be either up or down according to your choice.

Cause the program to display your name in some manner.

Figure 2 Graphic output from Exercise 1.



Exercise 2

Using Java and the game-math library named **GM2D02** , or using a different programming environment of your choice, write a program that draws x and y axes in a 2D Cartesian coordinate system as shown in [Figure 3](#).

Cause the origin of your reference frame be at the center of your drawing.

Cause the positive x direction be to the right.

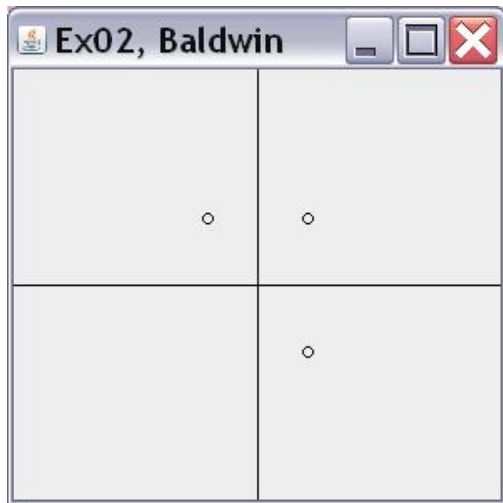
Cause the positive y direction be either up or down according to your choice.

Then draw small circles that represent the locations of points at the following coordinates in the reference frame as shown in [Figure 3](#).

- $x = -30, y = -40$
- $x = 30, y = -40$
- $x = 30, y = 40$

Cause the program to display your name in some manner.

Figure 3 Graphic output from Exercise 2.



Exercise 3

Using Java and the game-math library named **GM2D02** , or using a different programming environment of your choice, write a program that draws x and y axes in a 2D Cartesian coordinate system as shown in [Figure 4](#).

Cause the origin of your reference frame be at the center of your drawing.

Cause the positive x direction be to the right.

Cause the positive y direction be either up or down according to your choice.

Create six random values in the range from -128 to +127. Use those values as the x and y coordinates for three points in the 2D reference frame. *(The code fragment in [Listing 29](#) shows how to generate random values in the required range in Java.)*

Display the six values on a text screen labeled in such a way that it is possible to associate the values with the points as shown in [Figure 5](#).

Draw three small circles that represent the locations of the points.

Draw three line segments that connect the three points in pairs creating the shape of a triangle.

Cause the program to display your name in some manner.

Figure 4 Graphic output from Exercise 3.

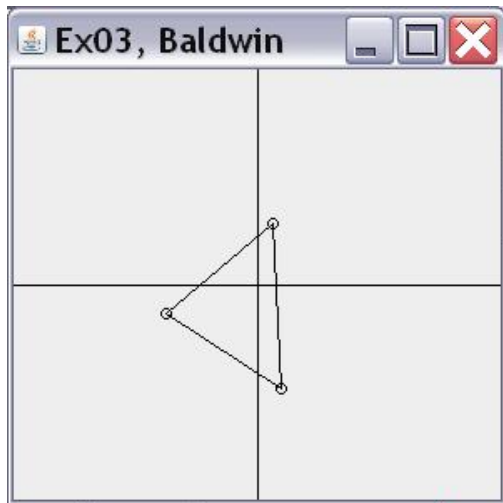


Figure 5 . Text output from Exercise 3.

```
point0X: -55  
point0Y: 17  
point1X: 9  
point1Y: -37  
point2X: 14  
point2Y: 62
```

Listing 29. Code for generating random values in Java.

Listing 29. Code for generating random values in Java.

```
//Define three randomly located points.  
Random generator = new Random(new Date().getTime());  
point0X = (byte)generator.nextInt();  
point0Y = (byte)generator.nextInt();  
point1X = (byte)generator.nextInt();  
point1Y = (byte)generator.nextInt();  
point2X = (byte)generator.nextInt();  
point2Y = (byte)generator.nextInt();
```

Exercise 4

Using Java and the game-math library named **GM2D02** , or using a different programming environment of your choice, write a program that draws three small circles that represent the locations of points at the following coordinates in a 2D reference frame.

- point0X = -30
- point0Y = 0
- point1X = 30
- point1Y = 40
- point2X = 30
- point2Y = -40

Then draw three line segments that extend from edge-to-edge of your drawing and intersect the three points in pairs as shown in [Figure 6](#).

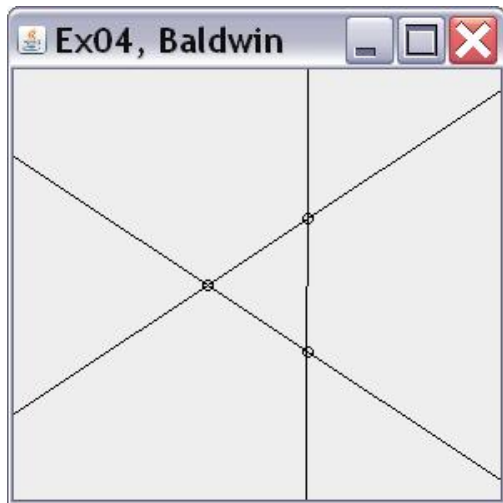
Cause the origin of your reference frame be at the center of your drawing.

Cause the positive x direction to be to the right.

Cause the positive y direction be either up or down according to your choice.

Cause the program to display your name in some manner.

Figure 6 Graphic output from Exercise 4.



Exercise 5

Using Java and the game-math library named **GM2D02**, or using a different programming environment of your choice, write a program that draws 24 vectors tail-to-tail in alternating colors of red, green, and blue as shown in [Figure 7](#).

The first vector is red.

The length of each vector is 100 pixels.

The direction of the first vector is +7.5 degrees relative to the horizontal.

The direction of each successive vector is increased by 15 degrees.

For example, the directions of the first four vectors are:

- 7.5
- 22.5
- 37.5
- 52.5

Draw the symbol of your choice to identify the head of each vector.

Draw the axes for a Cartesian coordinate system in the reference frame.

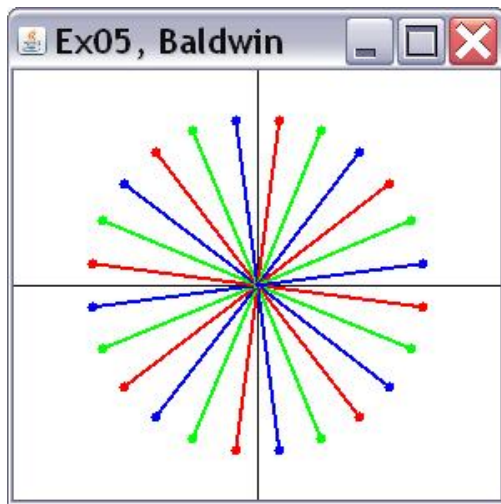
Cause the positive x direction to be to the right.

Cause the positive y direction be either up or down according to your choice.

Cause positive angles to be either clockwise or counter-clockwise according to your choice.

Cause the program to display your name in some manner.

Figure 7 Graphic output from Exercise 5.



Exercise 6

Using Java and the game-math library named **GM2D02** , or using a different programming environment of your choice, write a program that draws 24 vectors tail-to-head in alternating colors of red, green, and blue as shown in [Figure 8](#).

The first vector is red.

The length of each vector is approximately 17 pixels.

Draw the symbol of your choice to identify the head of each vector.

Draw the axes for a Cartesian coordinate system in the reference frame.

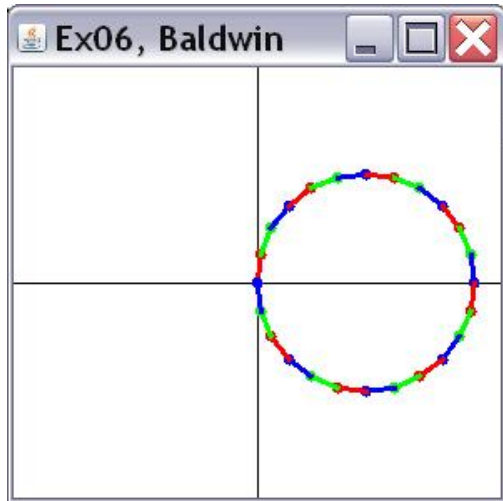
Cause the positive x direction to be to the right.

Cause the positive y direction be either up or down according to your choice.

Cause positive angles to be either clockwise or counter-clockwise according to your choice.

Cause the program to display your name in some manner.

Figure 8 Graphic output from Exercise 6.



Exercise 7

Using Java and the game-math library named **GM2D02** , or using a different programming environment of your choice, write a program that draws 24 vectors tail-to-head in alternating colors of red, green, and blue as shown in [Figure 9](#).

The first vector is red.

Begin with a length of 24 pixels for the first vector and reduce the length of each successive vector by one pixel.

Draw the symbol of your choice to identify the head of each vector.

Draw the axes for a Cartesian coordinate system in the reference frame.

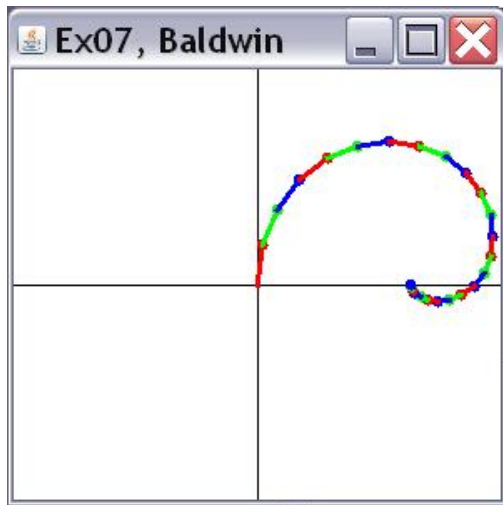
Cause the positive x direction to be to the right.

Cause the positive y direction be either up or down according to your choice.

Cause positive angles to be either clockwise or counter-clockwise according to your choice.

Cause the program to display your name in some manner.

Figure 9 Graphic output from Exercise 7.



-end-

Game0110r Review

This module contains review questions and answers keyed to the module titled [GAME 2302-0110: Updating the Game Math Library for Graphics](#).

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module contains review questions and answers keyed to the module titled [GAME 2302-0110: Updating the Game Math Library for Graphics](#).

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

A point is simply a location in space. It has no width, depth, or height. Therefore, it cannot be seen by the human eye, which means that we can't draw a point on the computer screen. However, it is possible to draw an object on the computer screen that indicates the location of the point.

True or False?

[Answer 1](#)

Question 2

A point in space exists only after a visual object is created to mark the location of the point.

[Answer 2](#)

Question 3

A point is a small black circle.

True or False?

[Answer 3](#)

Question 4

Kjell tells us that a line segment is the straight path between two points, and that it has no thickness. Since it has no thickness, a line cannot be seen by the human eye.

True or False?

[Answer 4](#)

Question 5

GM2D02.Point objects represent themselves as small black squares in a graphics context for the benefit of human observers.

True or False?

[Answer 5](#)

Question 6

GM2D02.Vector objects represent themselves as lines with arrow-heads at their heads in a graphics context for the benefit of human observers.

True or False?

[Answer 6](#)

Question 7

A **GM2D02.Vector** object is:

Select the correct answer

- A. An underlying data object
- B. A graphics object
- C. Both of the above
- D. None of the above

[Answer 7](#)

Question 8

All Java parameters are passed to methods by reference.

True or False?

[Answer 8](#)

Question 9

Java supports both primitive variables and reference variables.

True or False?

[Answer 9](#)

Question 10

The **GM2D02** library is an update of the earlier game-math library named **GM2D01** . This update added the following new capabilities:

Select the correct answer.

- A. Draw circles to represent **GM2D02.Point** objects.
- B. Draw visible lines to represent **GM2D02.Line** objects.
- C. Draw visible lines and circles to represent **GM2D02.Vector** objects.
- D. Call a **setData** method to change the values in a **GM2D02.ColMatrix** object.
- E. Call a **setData** method to change the values in a **GM2D02.Point** object.
- F. Call a **setData** method to change the values in a **GM2D02.Vector** object.
- G. Call a **setHead** method or a **setTail** method to change one of the points that defines a **GM2D02.Line** object.
- H. All of the above
- I. None of the above

[Answer 10](#)

Question 11

The updated constructors in the **GM2D02** library create and save clones of the **ColMatrix** and/or **Point** objects originally used to define the **Point** , **Vector** , and/or **Line** objects.

True or False?

[Answer 11](#)

Question 12

The **createImage** method belonging to a **JFrame** object can be used to create an off-screen image.

True or False?

[Answer 12](#)

Question 13

If the contents of an off-screen image are displayed on the screen, the units of width and height are measured in pixels.

True or False?

[Answer 13](#)

Question 14

The returned value of the **createImage** method is type **JFrame** .

True or False?

[Answer 14](#)

Question 15

The return type of the **getGraphics** methods is **Graphics** .

True or False?

[Answer 15](#)

Question 16

Objects instantiated from the **GM2D02.Point** and **GM2D02.Line** classes are not graphical objects.

True or False?

[Answer 16](#)

Question 17

Objects instantiated from the **GM2D02.Point** and **GM2D02.Line** classes are underlying data objects that are suitable for use in mathematical operations.

True or False?

[Answer 17](#)

Question 18

A call to the **draw** method of the **GM2D02.Line** class causes an object of the standard Java **Point2D.Double** class to be rendered onto the specified graphics context.

True or False?

[Answer 18](#)

Question 19

The **Line2D.Double** object is a graphical object and the **GM2D02.Line** is an underlying data object.

True or False?

[Answer 19](#)

Question 20

The **GM2D02.Line** class is a member of the standard Java library, whereas the **Line2D.Double** class is a member of the special game math library named **GM2D02** .

True or False?

[Answer 20](#)

Question 21

One of the problems with object-oriented programming is that objects don't know how to do useful things for themselves.

True or False?

[Answer 21](#)

Question 22

A call to the **draw** method of the **GM2D02.Point** class causes an object of the standard Java **Ellipse2D.Double** class to be rendered onto the specified graphics context.

True or False?

[Answer 22](#)

Question 23

The constructor for the Java standard **Ellipse2D.Double** class requires four incoming parameters that specify the coordinates of the upper-left corner of a rectangle followed by the width and the height of the rectangle.

True or False?

[Answer 23](#)

Question 24

An object of type **Ellipse2D.Double** represents an ellipse that is bounded by the four sides of a rectangle. If the rectangle is square, the ellipse becomes a square.

True or False?

[Answer 24](#)

Question 25

Kjell tells us that a vector is a geometrical object that has two properties: length and direction. He also tells us that a vector does not have a position.

True or False?

[Answer 25](#)

Question 26

The **draw** method of the **GM2D02.Vector** class requires three parameters:

- A reference to an object of the **GM2D02.Vector** class.
- A reference to the off-screen graphics context
 - on which the visual manifestation of the vector will be drawn.
- A reference to an object of the class **GM2D02.Point**
 - that will be used to determine the position on the off-screen image in which the visual manifestation will appear.

True or False?

[Answer 26](#)

Question 27

The visual manifestation of a vector can be placed anywhere in space, and one placement is just as correct as the next

True or False?

[Answer 27](#)

Question 28

Although the visual manifestation of a vector can be placed anywhere in space, and one placement is just as correct as the next, certain placements may be preferable to others in some cases so as to better represent the problem being modeled by the use of vectors.

True or False?

[Answer 28](#)

Question 29

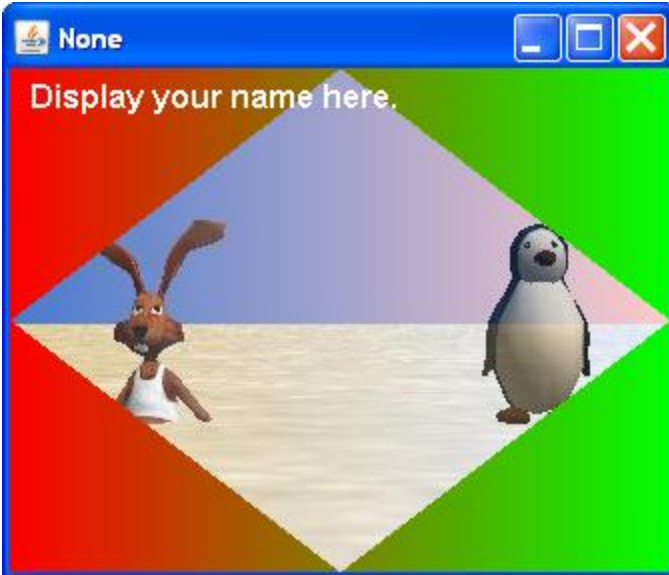
The rendering of a **GM2D02.Vector** object by the draw method of the **GM2D02.Vector** class draws a small circle to visually identify the head of the vector.

True or False?

[Answer 29](#)

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 29

True

[Back to Question 29](#)

Answer 28

True. One example is the graphical addition of vectors using the tail-to-head placement.

[Back to Question 28](#)

Answer 27

True, because according to Kjell, a vector doesn't have a position. Hence, there is nothing in the underlying data for a **GM2D02.Vector** object that

specifies a position.

[Back to Question 27](#)

Answer 26

False. The **draw** method of the **GM2D02.Vector** class requires two parameters:

- A reference to the off-screen graphics context
 - on which the visual manifestation of the vector will be drawn.
- A reference to an object of the class **GM2D02.Point**
 - that will be used to determine the position on the off-screen image in which the visual manifestation will appear.

[Back to Question 26](#)

Answer 25

True

[Back to Question 25](#)

Answer 24

False. An object of type **Ellipse2D.Double** represents an ellipse that is bounded by the four sides of a rectangle. If the rectangle is square, the ellipse becomes a *circle* .

[Back to Question 24](#)

Answer 23

True

[Back to Question 23](#)

Answer 22

True

[Back to Question 22](#)

Answer 21

False. One of the hallmarks of object-oriented programming is that objects do know how to do useful things for themselves.

[Back to Question 21](#)

Answer 20

False. The **Line2D.Double** class is a member of the standard Java library, whereas the **GM2D02.Line** class is a member of the special game math library named **GM2D02**.

[Back to Question 20](#)

Answer 19

True

[Back to Question 19](#)

Answer 18

False. A call to the **draw** method of the **GM2D02.Line** class causes an object of the standard Java **Line2D.Double** class to be rendered onto the specified graphics context.

[Back to Question 18](#)

Answer 17

True

[Back to Question 17](#)

Answer 16

True

[Back to Question 16](#)

Answer 15

True

[Back to Question 15](#)

Answer 14

False. The returned value of the **createImage** method is type **Image** .

[Back to Question 14](#)

Answer 13

True

[Back to Question 13](#)

Answer 12

True

[Back to Question 12](#)

Answer 11

True

[Back to Question 11](#)

Answer 10

H. All of the above

[Back to Question 10](#)

Answer 9

True

[Back to Question 9](#)

Answer 8

False. All Java parameters are passed to methods by value.

[Back to Question 8](#)

Answer 7

A. An underlying data object

[Back to Question 7](#)

Answer 6

False. **GM2D02.Vector** objects represent themselves as lines with small circles at their heads in a graphics context for the benefit of human observers.

[Back to Question 6](#)

Answer 5

False. **GM2D02.Point** objects represent themselves as small black circles in a graphics context for the benefit of human observers.

[Back to Question 5](#)

Answer 4

True

[Back to Question 4](#)

Answer 3

False. A point is simply a location in space. It has no width, depth, or height. Therefore, it cannot be seen by the human eye, which means that we can't draw a point on the computer screen. However, it is possible to draw an object on the computer screen that indicates the location of the point.

[Back to Question 3](#)

Answer 2

False. Points in space exist regardless of whether or not visual objects are created to mark the locations of the points.

[Back to Question 2](#)

Answer 1

True

[Back to Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0110r Review: Updating the Game Math Library for Graphics
- File: Game0110r.htm
- Published: 12/31/12
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0115: Working with Column Matrices, Points, and Vectors

Learn how to compare column matrices for equality, compare two points for equality, compare two vectors for equality, add one column matrix to another, subtract one column matrix from another, and get a displacement vector from one point to another.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The game-math library named GM2D03](#)
 - [The sample program named ColMatrixEquals01](#)
 - [The sample program named DisplacementVector01](#)
 - [The sample program named ColMatrixAddSubtract01](#)
- [Documentation for the GM2D03 library](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)
 - [Exercise 3](#)

Preface

This module is one in a series of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, you learned how to update the game-math library to provide new capabilities including the addition of graphics and **set** methods for column matrices, points, vectors, and lines. You also learned how to draw on off-screen images.

What you will learn

In this module, you will learn how to compare column matrices for equality, compare two points for equality, compare two vectors for equality, add one column matrix to another, subtract one column matrix from another, and get a displacement vector from one point to another.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output from the program named ColMatrixEquals01.
- [Figure 2](#). Screen output from the program named DisplacementVector01.
- [Figure 3](#). Screen output from the program named ColMatrixAddSubtract01.
- [Figure 4](#). Text output from Exercise 1.
- [Figure 5](#). Text output from Exercise 2.
- [Figure 6](#). Text output from Exercise 3.

Listings

- [Listing 1](#). Overridden equals method of the GM2D03.ColMatrix class.
- [Listing 2](#). Overridden equals method of the GM2D03.Point class.
- [Listing 3](#). The getColMatrix method of the GM2D03.Point class.
- [Listing 4](#). Beginning of the ColMatrixEquals01 class.
- [Listing 5](#). Remainder of the ColMatrixEquals01 class.
- [Listing 6](#). The getDisplacementVector method of the GM2D03.Point class.
- [Listing 7](#). The program named DisplacementVector01.
- [Listing 8](#). Source code for the program named ColMatrixAddSubtract01.
- [Listing 9](#). Source code for the add method of the GM2D03.ColMatrix class.
- [Listing 10](#). Source code for the subtract method of the GM2D03.ColMatrix class.
- [Listing 11](#). Source code for game-math library named GM2D03.
- [Listing 12](#). Source code for the program named ColMatrixEquals01.
- [Listing 13](#). Source code for the program named DisplacementVector01.
- [Listing 14](#). Source code for the program named ColMatrixAddSubtract01.

Preview

As mentioned earlier, in this module you will learn how to:

- Compare two column matrices for equality.
- Compare two points for equality.
- Compare two vectors for equality.
- Add one column matrix to another.
- Subtract one column matrix from another.
- Get a displacement vector from one point to another.

To assist you in this quest, I will present and explain modifications that were made to update the game-math library that you learned about in the previous two modules. In addition, I will present and explain three sample programs that illustrate the new features of the game-math library.

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

Much of the code in the library remains unchanged. I explained that code in previous modules and I won't repeat that explanation in this module. Rather, in this module, I will concentrate on explaining the modifications that I made to the library.

The game-math library named GM2D03

A complete listing of the library program is provided in [Listing 11](#) near the end of the module.

This update added the following new capabilities:

- Compare two **ColMatrix** objects for equality by implementing Kjell's rules for equality given in his Chapter 1, topic "*Column Matrix Equality*." The equality test does not test for absolute equality. Rather, It compares the values stored in two matrices and returns true if the values are *equal or almost equal* and returns false otherwise.
- Get a reference to the **ColMatrix** object that defines a **Point** object.
- Compare two **Point** objects for equality based on a comparison of the **ColMatrix** objects that define them.
- Get a reference to the **ColMatrix** object that defines a **Vector** object.
- Compare two **Vector** objects for equality based on a comparison of the **ColMatrix** objects that define them.

- Add one **ColMatrix** object to a second **ColMatrix** object, returning a **ColMatrix** object.
- Subtract one **ColMatrix** object from a second **ColMatrix** object, returning a **ColMatrix** object.
- Get a displacement vector from one **Point** object to a second **Point** object. The vector points from the object on which the **getDisplacementVector** method is called to the object passed as a parameter to the method.

I will explain these updates in conjunction with the discussions of the programs that follow.

The sample program named ColMatrixEquals01

A complete listing of this program is provided in [Listing 12](#) near the end of the module. I will explain the program in fragments. In selecting the portions of the program that I will explain, I will skip over material that is very similar to code that I have previously explained.

The purpose of this program is to confirm the behavior of the **equals** methods of the **GM2D03.ColMatrix** , **Point** , and **Vector** classes.

Overridden *equals* method of the GM2D03.ColMatrix class

I will begin by explaining some of the **equals** methods in the updated **GM2D03** game-math library.

The first fragment in [Listing 1](#) shows the new **equals** method of the **ColMatrix** class.

Listing 1 . Overridden equals method of the GM2D03.ColMatrix class.

Listing 1 . Overridden equals method of the GM2D03.ColMatrix class.

```
public boolean equals(Object obj){
    if(obj instanceof GM2D03.ColMatrix &&
        Math.abs(((GM2D03.ColMatrix)obj).getData(0) -
                    getData(0)) <= 0.00001
    &&
        Math.abs(((GM2D03.ColMatrix)obj).getData(1) -
                    getData(1)) <= 0.00001)
    {
        return true;
    }else{
        return false;
    }//end else
}

//end overridden equals method
```

This method overrides the **equals** method inherited from the **Object** class. It compares the **double** values stored in two matrices and returns true if the values are *equal or almost equal* and returns false otherwise.

An inherent problem when comparing doubles and floats

There is always a problem when comparing two **double** or **float** values for equality. If you perform a series of computations twice, using a different computational order for each set of computations, you are likely to end up with two values that are not absolutely equal even if they should be equal. Arithmetic inaccuracies along the way may cause the two results to differ ever so slightly. However, they may be equal from a practical viewpoint.

Note: An example of the problem:

See [Figure 3](#) where the a value that should be 0.0 is actually given by
-1.7763568394002505E-15

Therefore, when comparing two **double** or **float** values for equality, it is customary to subtract one from the other, convert the difference to an absolute value, and compare that absolute value with an arbitrarily small positive value. If the difference is less than that the test value, the two original values are declared to be equal. Otherwise, they are declared to be unequal.

This is the logic that is implemented in [Listing 1](#), which shouldn't require further explanation.

Overridden *equals* method of the GM2D03.Point class

[Listing 2](#) presents the overridden equals method of the GM2D03.Point class.

Listing 2 . Overridden equals method of the GM2D03.Point class.

```
public boolean equals(Object obj){
    if(point.equals(((GM2D03.Point)obj).
                                getColMatrix()))
{
    return true;
}else{
    return false;
} //end else

} //end overridden equals method
```

This method also overrides the **equals** method inherited from the **Object** class. It compares the values stored in the **ColMatrix** objects that define two **Point** objects and returns true if they are equal and false otherwise.

One possible point of confusion

The one thing that can be confusing in [Listing 2](#) has to do with the variable named **point** . This is a reference to an object of type **ColMatrix** that defines the location of the **Point** object. [Listing 2](#) calls another new method named **getColMatrix** to get access to the **ColMatrix** object that defines the incoming **Point** object, and uses that object for the comparison. In effect, this method actually compares two objects of the **ColMatrix** class by calling the method that I explained in [Listing 1](#). This illustrates the advantages of building up the library classes using objects of the fundamental **ColMatrix** class.

The getColMatrix method of the GM2D03.Point class

This extremely simple new method, which is called by the code in [Listing 2](#), is presented in [Listing 3](#).

Listing 3 . The getColMatrix method of the GM2D03.Point class.

```
//Returns a reference to the ColMatrix object that
// defines this Point object.
public GM2D03.ColMatrix getColMatrix(){
    return point;
} //end getColMatrix
```

[Listing 3](#) shouldn't require any explanation beyond the embedded comments.

Overridden *equals* method of the GM2D03.Vector class

The overridden **equals** method of the **GM2D03.Vector** class is essentially the same as the code shown for the **Point** class in [Listing 2](#) so I won't bother to show and explain it. You can view this new method in [Listing 11](#).

Beginning of the ColMatrixEquals01 program class

At this point, I will put the discussion of the updated **GM2D03** library on hold and explain the **ColMatrixEquals01** program.

[Listing 4](#) presents the beginning of the **ColMatrixEquals01** program class including the beginning of the **main** method.

Listing 4 . Beginning of the ColMatrixEquals01 class.

Listing 4 . Beginning of the ColMatrixEquals01 class.

```
public class ColMatrixEquals01{
    public static void main(String[] args){
        GM2D03.ColMatrix matA =
                                new
GM2D03.ColMatrix(1.5, -2.6);
        GM2D03.ColMatrix matB =
                                new
GM2D03.ColMatrix(1.5, -2.6);
        GM2D03.ColMatrix matC =
                                new
GM2D03.ColMatrix(1.500001, -2.600001);
        GM2D03.ColMatrix matD =
                                new
GM2D03.ColMatrix(1.50001, -2.60001);
        System.out.println(matA.equals(matA));
        System.out.println(matA.equals(matB));
        System.out.println(matA.equals(matC));
        System.out.println(matA.equals(matD));
    }
}
```

[Listing 4](#) instantiates four different **ColMatrix** objects and then compares them in different ways, displaying the results of the comparisons on the command-line screen.

Screen output from the program named ColMatrixEquals01

The first four lines of text in [Figure 1](#) were produced by the code in Listing 4. (The remaining output shown in [Figure 1](#) was produced by the code in [Listing 5](#), which I will explain shortly.)

Figure 1 . Screen output from the program named ColMatrixEquals01.

Figure 1 . Screen output from the program named ColMatrixEquals01.

```
true
true
true
false

true
true
false

true
true
false
```

Because of the simplicity of the code in [Listing 4](#), you shouldn't need any help in understanding why the code in [Listing 4](#) produced the first four lines of output in [Figure 1](#).

The third and fourth lines of output in [Figure 1](#) are the result of comparing two matrices whose values are almost equal but not absolutely equal.

Remainder of the ColMatrixEquals01 class

The remainder of the program named **ColMatrixEquals01** is shown in [Listing 5](#). The output produced by this code is shown in the last six lines of text in [Figure 1](#).

Listing 5 . Remainder of the ColMatrixEquals01 class.

Listing 5 . Remainder of the ColMatrixEquals01 class.

```
        GM2D03.Point pointA = new GM2D03.Point(
            new
GM2D03.ColMatrix(15.6, -10.11));
        GM2D03.Point pointB = new GM2D03.Point(
            new
GM2D03.ColMatrix(15.6, -10.11));
        GM2D03.Point pointC = new GM2D03.Point(
            new
GM2D03.ColMatrix(-15.6, 10.11));
        System.out.println(/*Blank line*/);
        System.out.println(pointA.equals(pointA));
        System.out.println(pointA.equals(pointB));
        System.out.println(pointA.equals(pointC));

        GM2D03.Vector vecA = new GM2D03.Vector(
            new
GM2D03.ColMatrix(15.6, -10.11));
        GM2D03.Vector vecB = new GM2D03.Vector(
            new
GM2D03.ColMatrix(15.6, -10.11));
        GM2D03.Vector vecC = new GM2D03.Vector(
            new
GM2D03.ColMatrix(-15.6, 10.11));
        System.out.println(/*Blank line*/);
        System.out.println(vecA.equals(vecA));
        System.out.println(vecA.equals(vecB));
        System.out.println(vecA.equals(vecC));

    } //end main
} //end ColMatrixEquals01 class
```

Once again, the code in [Listing 5](#) is very straightforward and shouldn't require further explanation. The screen output shown in [Figure 1](#) verifies that the library methods called by this program behave appropriately.

The sample program named DisplacementVector01

A *displacement vector* describes the distance and direction that you would have to move to get from one point in space to another point in space.

The `getDisplacementVector` method of the `GM2D03.Point` class

Returning to the discussion of the updated `GM2D03` library, [Listing 6](#) presents the new `getDisplacementVector` method of the `GM2D03.Point` class

Listing 6 . The `getDisplacementVector` method of the `GM2D03.Point` class.

```
public GM2D03.Vector getDisplacementVector(
                                GM2D03.Point point)
{
    return new GM2D03.Vector(new GM2D03.ColMatrix(
                                point.getData(0)-getData(0),
                                point.getData(1)-
                                getData(1)));
} //end getDisplacementVector
```

This method gets and returns a displacement vector from one **Point** object to a second **Point** object and returns the result as a reference to a new object of the class `GM2D03.Vector` .

The direction of the vector

The displacement vector points from the **Point** object on which the method is called to the **Point** object passed as a parameter to the method. Kjell describes the component parts of the new vector as the distance you would have to walk, first along the x-axis and then along the y-axis to get from the first point to the second point. Of course, you could take the short cut and walk directly from the first point to the second point but that's often not how we do it in programming.

The code in [Listing 6](#) is straightforward and shouldn't require further explanation.

The program named `DisplacementVector01`

Once again, I will put the discussion of the updated `GM2D03` library on hold and explain the program named `DisplacementVector01` .

[Listing 7](#) shows the program named **DisplacementVector01** in its entirety.

(For convenience, a second copy of this program is provided in [Listing 13](#) near the end of the module along with the other three programs discussed in this module.)

Listing 7 . The program named DisplacementVector01.

```
public class DisplacementVector01{
    public static void main(String[] args){
        GM2D03.Point pointA = new GM2D03.Point(
            new
GM2D03.ColMatrix(6.5, -9.7));
        GM2D03.Point pointB = new GM2D03.Point(
            new
GM2D03.ColMatrix(-6.0, 9.0));

        System.out.println(pointA.getDisplacementVector(
pointB));
        System.out.println(pointB.getDisplacementVector(
pointA));

        }//end main
    }//end DisplacementVector01 class
```

The purpose of this program is to confirm the behavior of the **getDisplacementVector** method of the **GM2D03.Point** class. The screen output shown in [Figure 2](#) provides that confirmation.

Screen output from the program named DisplacementVector01

The screen output from the program named **DisplacementVector01** is shown in [Figure 2](#).

Figure 2 . Screen output from the program named DisplacementVector01.

```
-12.5, 18.7  
12.5, -18.7
```

You should be able to correlate the results shown in [Figure 2](#) with the code in [Listing 6](#) and [Listing 7](#) without further explanation.

Very simple methods

By now you may be thinking that the code in the game-math library is very simple and easy to understand. I hope that is the case. I went to great lengths to modularize the library into a set of simple and easily understood methods. Taken as a whole, however, the library is becoming quite powerful, and will become even more powerful as we progress through additional modules in this collection.

The sample program named ColMatrixAddSubtract01

The source code for this program is shown in its entirety in [Listing 8](#).

(The source code is also provided in [Listing 14](#) near the end of the module for convenience.)

Listing 8 . Source code for the program named ColMatrixAddSubtract01.

Listing 8 . Source code for the program named ColMatrixAddSubtract01.

```
public class ColMatrixAddSubtract01{
    public static void main(String[] args){

        GM2D03.ColMatrix matrixA =
            new
GM2D03.ColMatrix(3.14, -6.01);
        GM2D03.ColMatrix matrixB =
            new
GM2D03.ColMatrix(-14.0, -12.2);

        GM2D03.ColMatrix matrixC = matrixA.add(matrixB);
        GM2D03.ColMatrix matrixD = matrixC.subtract(matrixA);
        GM2D03.ColMatrix matrixE = matrixD.subtract(matrixB);

        System.out.println(matrixC);
        System.out.println(matrixD);
        System.out.println(matrixE);
    }//end main
}

} //end class ColMatrixAddSubtract01
```

The purpose of this program is to confirm the behavior of the new **add** and **subtract** methods of the GM2D03.ColMatrix class.

Source code for the add method of the GM2D03.ColMatrix class

Returning once more to a discussion of the updated **GM2D03** library, the source code for this method is shown in [Listing 9](#). The method adds one **ColMatrix** object to another **ColMatrix** object, returning a **ColMatrix** object. As you should have learned from your studies of the Kjell tutorial, the order in which the two objects are added doesn't matter. The result is the same either way.

Listing 9 . Source code for the add method of the GM2D03.ColMatrix class.

Listing 9 . Source code for the add method of the GM2D03.ColMatrix class.

```
public GM2D03.ColMatrix add(GM2D03.ColMatrix matrix){
    return new GM2D03.ColMatrix(
        getData(0)+matrix.getData(0),
        getData(1)+matrix.getData(1));
} //end add
```

The code in the method is straightforward and shouldn't require an explanation.

Source code for the subtract method of the GM2D03.ColMatrix class

Continuing with the discussion of the updated **GM2D03** library, the source code for this method is shown in [Listing 10](#). This method subtracts one **ColMatrix** object from another **ColMatrix** object, returning a **ColMatrix** object. Also as you should have learned from the Kjell tutorial, in this case, the order of the operation does matter. The object that is received as an incoming parameter is subtracted from the object on which the method is called. Reversing the order of operations produces different results.

Listing 10 . Source code for the subtract method of the GM2D03.ColMatrix class.

```
public GM2D03.ColMatrix subtract(
    GM2D03.ColMatrix matrix)
{
    return new GM2D03.ColMatrix(
        getData(0)-matrix.getData(0),
        getData(1)-
        matrix.getData(1));
} //end subtract
```

Once again, the code in the method is straightforward and shouldn't require an explanation.

Program output for program named ColMatrixAddSubtract01

Now, let's get back to the program named **ColMatrixAddSubtract01** that is shown in [Listing 8](#). This program produces the output shown in [Figure 3](#).

Figure 3 . Screen output from the program named ColMatrixAddSubtract01.

```
-10.86, -18.21  
-14.0, -12.2000000000000001  
0.0, -1.7763568394002505E-15
```

By now, you should have no difficulty understanding how the output shown in [Figure 3](#) was produced by the code shown in [Listing 8](#). Note, however, that the last value on the third row should be 0.0 instead of the extremely small value shown. This is an example of an [inherent](#) problem having to do with comparing **double** or **float** values with other **double** or **float** values for absolute equality.

Documentation for the GM2D03 library

Click [here](#) to download a zip file containing standard javadoc documentation for the library named **GM2D03**. Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text (see [Listing 11](#) and the explanations given above), the documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

Homework assignment

Your homework assignment for this module was to study Kjell's tutorial through Chapter 2 - *Column and Row Matrix Addition*.

The homework assignment for the next module is to study the Kjell tutorial through Chapter 3 - *Vector Addition*.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 12](#), [Listing 13](#), and [Listing 14](#). Compile the code and execute it in conjunction with the game-math library provided in [Listing 11](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to compare column matrices for equality, compare two points for equality, compare two vectors for equality, add one column matrix to another, subtract one column matrix from another, and get a displacement vector from one point to another.

What's next?

The next module in the collection will show one way for you to visualize column matrices in graphical form.

In the module following that one, you will learn:

- How to add two or more vectors.
- About the head-to-tail rule in vector addition.
- About the vector addition parallelogram.
- About the relationship between the length of the sum of vectors and the sum of the lengths of the vectors.
- How to add a vector to a point.
- How to get the length of a vector.
- How to represent an object in different coordinate frames.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME2302-0115: Working with Column Matrices, Points, and Vectors
- File: Game0115.htm
- Published: 10/14/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 11](#) through [Listing 14](#) below.

Listing 11 . Source code for game-math library named GM2D03.

```
/*GM2D03.java  
Copyright 2008, R.G.Baldwin  
Revised 02/10/08
```

The name GM2Dnn is an abbreviation for GameMath2Dnn.

See the file named GM2D01.java for a general description of this game-math library file. This file is an update of GM2D02.

This update added the following new capabilities:

Compare two ColMatrix objects for equality by implementing Kjell's rules for equality given in his Chapter 1, topic "Column Matrix Equality." The equality test does not test for absolute equality. Rather, It compares the values stored in two matrices and returns true if the values are equal or almost equal and returns false otherwise.

Get a reference to the ColMatrix object that defines a Point object.

Compare two Point objects for equality based on a comparison of the ColMatrix objects that define them.

Get a reference to the ColMatrix object that defines a Vector object.

Compare two Vector objects for equality based on a comparison of the ColMatrix objects that define them.

Add one ColMatrix object to a second ColMatrix object, returning a ColMatrix object.

Subtract one ColMatrix object from a second ColMatrix object, returning a ColMatrix object.

Get a displacement vector from one Point object to a second Point object. The vector points from the object on which the getDisplacementVector method is called to the object passed as a parameter to the method.

Tested using JDK 1.6 under WinXP.

*****/

```
import java.awt.geom.*;
import java.awt.*;
```

```
public class GM2D03{
```

```
    //An object of this class represents a 2D column matrix.
    // An object of this class is the fundamental building
    // block for several of the other classes in the
    // library.
```

```
    public static class ColMatrix{
        double[] data = new double[2];
```

```
        public ColMatrix(double data0,double data1){
```

```

    data[0] = data0;
    data[1] = data1;
} //end constructor
//-----//

public String toString(){
    return data[0] + "," + data[1];
} //end overridden toString method
//-----//

public double getData(int index){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        return data[index];
    } //end else
} //end getData method
//-----//

public void setData(int index, double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        this.data[index] = data;
    } //end else
} //end setData method
//-----//

//This method overrides the equals method inherited
// from the class named Object. It compares the values
// stored in two matrices and returns true if the
// values are equal or almost equal and returns false
// otherwise.
public boolean equals(Object obj){
    if(obj instanceof GM2D03.ColMatrix &&
        Math.abs(((GM2D03.ColMatrix)obj).getData(0) -
                    getData(0)) <= 0.00001 &&
        Math.abs(((GM2D03.ColMatrix)obj).getData(1) -
                    getData(1)) <= 0.00001){
        return true;
    }else{
        return false;
    } //end else
} //end overridden equals method

```

```

//-----//

//Adds one ColMatrix object to another ColMatrix
// object, returning a ColMatrix object.
public GM2D03.ColMatrix add(GM2D03.ColMatrix matrix){
    return new GM2D03.ColMatrix(
        getData(0)+matrix.getData(0),
        getData(1)+matrix.getData(1));
} //end add
//-----//

//Subtracts one ColMatrix object from another
// ColMatrix object, returning a ColMatrix object. The
// object that is received as an incoming parameter
// is subtracted from the object on which the method
// is called.
public GM2D03.ColMatrix subtract(
        GM2D03.ColMatrix matrix){
    return new GM2D03.ColMatrix(
        getData(0)-matrix.getData(0),
        getData(1)-matrix.getData(1));
} //end subtract
//-----//
} //end class ColMatrix
//=====//

public static class Point{
    GM2D03.ColMatrix point;

    public Point(GM2D03.ColMatrix point){ //constructor
        //Create and save a clone of the ColMatrix object
        // used to define the point to prevent the point
        // from being corrupted by a later change in the
        // values stored in the original ColVector object
        // through use of its set method.
        this.point =
            new ColMatrix(point.getData(0),point.getData(1));
    } //end constructor
    //-----//

    public String toString(){
        return point.getData(0) + "," + point.getData(1);
    } //end toString
    //-----//

```



```

public double getData(int index){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        return point.getData(index);
    }//end else
}//end getData
//-----//

public void setData(int index,double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        point.setData(index,data);
    }//end else
}//end setData
//-----//

//This method draws a small circle around the location
// of the point on the specified graphics context.
public void draw(Graphics2D g2D){
    Ellipse2D.Double circle =
        new Ellipse2D.Double(getData(0)-3,
                               getData(1)-3,
                               6,
                               6);

    g2D.draw(circle);
}//end draw
//-----//

//Returns a reference to the ColMatrix object that
// defines this Point object.
public GM2D03.ColMatrix getColMatrix(){
    return point;
}//end getColMatrix
//-----//

//This method overrides the equals method inherited
// from the class named Object. It compares the values
// stored in the ColMatrix objects that define two
// Point objects and returns true if they are equal
// and false otherwise.
public boolean equals(Object obj){
    if(point.equals(((GM2D03.Point)obj).
                    getColMatrix())){

```

```

        return true;
    }else{
        return false;
    }//end else

} //end overridden equals method
//-----//

//Gets a displacement vector from one Point object to
// a second Point object. The vector points from the
// object on which the method is called to the object
// passed as a parameter to the method. Kjell
// describes this as the distance you would have to
// walk along the x and then the y axes to get from
// the first point to the second point.
public GM2D03.Vector getDisplacementVector(
                                GM2D03.Point point){
    return new GM2D03.Vector(new GM2D03.ColMatrix(
                                point.getData(0)-getData(0),
                                point.getData(1)-getData(1)));
} //end getDisplacementVector
//-----//
} //end class Point
//=====//

public static class Vector{
    GM2D03.ColMatrix vector;

    public Vector(GM2D03.ColMatrix vector){ //constructor
        //Create and save a clone of the ColMatrix object
        // used to define the vector to prevent the vector
        // from being corrupted by a later change in the
        // values stored in the original ColVector object.
        this.vector = new ColMatrix(
                                vector.getData(0),vector.getData(1));
    } //end constructor
    //-----//

    public String toString(){
        return vector.getData(0) + "," + vector.getData(1);
    } //end toString
    //-----//

    public double getData(int index){
        if((index < 0) || (index > 1)){

```

```

        throw new IndexOutOfBoundsException();
    }else{
        return vector.getData(index);
    }//end else
} //end getData
//-----//

public void setData(int index,double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        vector.setData(index,data);
    }//end else
} //end setData
//-----//

//This method draws a vector on the specified graphics
// context, with the tail of the vector located at a
// specified point, and with a small circle at the
// head.
public void draw(Graphics2D g2D,GM2D03.Point tail){
    Line2D.Double line = new Line2D.Double(
        tail.getData(0),
        tail.getData(1),
        tail.getData(0)+vector.getData(0),
        tail.getData(1)+vector.getData(1));

    //Draw a small circle to identify the head.
    Ellipse2D.Double circle = new Ellipse2D.Double(
        tail.getData(0)+vector.getData(0)-2,
        tail.getData(1)+vector.getData(1)-2,
        4,
        4);
    g2D.draw(circle);
    g2D.draw(line);
} //end draw
//-----//

//Returns a reference to the ColMatrix object that
// defines this Vector object.
public GM2D03.ColMatrix getColMatrix(){
    return vector;
} //end getColMatrix
//-----//

```

```

//This method overrides the equals method inherited
// from the class named Object. It compares the values
// stored in the ColMatrix objects that define two
// Vector objects and returns true if they are equal
// and false otherwise.
public boolean equals(Object obj){
    if(vector.equals((
        (GM2D03.Vector)obj).getColMatrix())){
        return true;
    }else{
        return false;
    }//end else

} //end overridden equals method
//-----//
} //end class Vector
//=====//

//A line is defined by two points. One is called the
// tail and the other is called the head.
public static class Line{
    GM2D03.Point[] line = new GM2D03.Point[2];

    public Line(GM2D03.Point tail,GM2D03.Point head){
        //Create and save clones of the points used to
        // define the line to prevent the line from being
        // corrupted by a later change in the coordinate
        // values of the points.
        this.line[0] = new Point(new GM2D03.ColMatrix(
            tail.getData(0),tail.getData(1)));
        this.line[1] = new Point(new GM2D03.ColMatrix(
            head.getData(0),head.getData(1)));
    } //end constructor
    //-----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ", "
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + ", "
            + line[1].getData(1);
    } //end toString
    //-----//

    public GM2D03.Point getTail(){
        return line[0];
    }

```

```

} //end getTail
//-----//

public GM2D03.Point getHead(){
    return line[1];
} //end getHead
//-----//

public void setTail(GM2D03.Point newPoint){
    //Create and save a clone of the new point to
    // prevent the line from being corrupted by a
    // later change in the coordinate values of the
    // point.
    this.line[0] = new Point(new GM2D03.ColMatrix(
        newPoint.getData(0), newPoint.getData(1)));
} //end setTail
//-----//

public void setHead(GM2D03.Point newPoint){
    //Create and save a clone of the new point to
    // prevent the line from being corrupted by a
    // later change in the coordinate values of the
    // point.
    this.line[1] = new Point(new GM2D03.ColMatrix(
        newPoint.getData(0), newPoint.getData(1)));
} //end setHead
//-----//

public void draw(Graphics2D g2D){
    Line2D.Double line = new Line2D.Double(
        getTail().getData(0),
        getTail().getData(1),
        getHead().getData(0),
        getHead().getData(1));

    g2D.draw(line);
} //end draw
//-----//
} //end class Line
//=====//

} //end class GM2D03

```

Listing 12 . Source code for the program named ColMatrixEquals01.

```
/*ColMatrixEquals01.java
Copyright 2008, R.G.Baldwin
Revised 02/08/08
```

The purpose of this program is to confirm the behavior of the equals methods of the GM2D03.ColMatrix, Point, and Vector classes.

Tested using JDK 1.6 under WinXP.

```
*****/
```

```
public class ColMatrixEquals01{
    public static void main(String[] args){
        GM2D03.ColMatrix matA =
            new GM2D03.ColMatrix(1.5,-2.6);
        GM2D03.ColMatrix matB =
            new GM2D03.ColMatrix(1.5,-2.6);
        GM2D03.ColMatrix matC =
            new GM2D03.ColMatrix(1.500001,-2.600001);
        GM2D03.ColMatrix matD =
            new GM2D03.ColMatrix(1.50001,-2.60001);
        System.out.println(matA.equals(matA));
        System.out.println(matA.equals(matB));
        System.out.println(matA.equals(matC));
        System.out.println(matA.equals(matD));

        GM2D03.Point pointA = new GM2D03.Point(
            new GM2D03.ColMatrix(15.6,-10.11));
        GM2D03.Point pointB = new GM2D03.Point(
            new GM2D03.ColMatrix(15.6,-10.11));
        GM2D03.Point pointC = new GM2D03.Point(
            new GM2D03.ColMatrix(-15.6,10.11));
        System.out.println(/*Blank line*/);
        System.out.println(pointA.equals(pointA));
        System.out.println(pointA.equals(pointB));
        System.out.println(pointA.equals(pointC));

        GM2D03.Vector vecA = new GM2D03.Vector(
            new GM2D03.ColMatrix(15.6,-10.11));
        GM2D03.Vector vecB = new GM2D03.Vector(
            new GM2D03.ColMatrix(15.6,-10.11));
        GM2D03.Vector vecC = new GM2D03.Vector(
            new GM2D03.ColMatrix(-15.6,10.11));
        System.out.println(/*Blank line*/);
        System.out.println(vecA.equals(vecA));
```

```

        System.out.println(vecA.equals(vecB));
        System.out.println(vecA.equals(vecC));

    }//end main
}//end ColMatrixEquals01 class

```

Listing 13 . Source code for the program named DisplacementVector01.

```

/*DisplacementVector01.java
Copyright 2008, R.G.Baldwin
Revised 02/08/08

```

The purpose of this program is to confirm the behavior of the getDisplacementVector method of the GM2D03.Point class.

Tested using JDK 1.6 under WinXP.

```

*****/

public class DisplacementVector01{
    public static void main(String[] args){
        GM2D03.Point pointA = new GM2D03.Point(
                                new GM2D03.ColMatrix(6.5,-9.7));
        GM2D03.Point pointB = new GM2D03.Point(
                                new GM2D03.ColMatrix(-6.0,9.0));

        System.out.println(pointA.getDisplacementVector(
                                pointB));
        System.out.println(pointB.getDisplacementVector(
                                pointA));

    }//end main
}//end DisplacementVector01 class

```

Listing 14 . Source code for the program named ColMatrixAddSubtract01.

```
/*ColMatrixAddSubtract01.java  
Copyright 2008, R.G.Baldwin  
Revised 02/08/08
```

The purpose of this program is to confirm the behavior of the add and subtract methods of the GM2D03.ColMatrix class.

Tested using JDK 1.6 under WinXP.

```
*****/
```

```
public class ColMatrixAddSubtract01{  
    public static void main(String[] args){  
  
        GM2D03.ColMatrix matrixA =  
            new GM2D03.ColMatrix(3.14, -6.01);  
        GM2D03.ColMatrix matrixB =  
            new GM2D03.ColMatrix(-14.0, -12.2);  
  
        GM2D03.ColMatrix matrixC = matrixA.add(matrixB);  
        GM2D03.ColMatrix matrixD = matrixC.subtract(matrixA);  
        GM2D03.ColMatrix matrixE = matrixD.subtract(matrixB);  
  
        System.out.println(matrixC);  
        System.out.println(matrixD);  
        System.out.println(matrixE);  
    }//end main  
  
}//end class ColMatrixAddSubtract01
```

Exercises

Exercise 1

Using Java and the game-math library named **GM2D03** , or using a different programming environment of your choice, write a program that creates four column matrix objects using the following names and values:

- matA = (1.5,-2.6)

- `matB = (1.5,-2.6)`
- `matC = (8.5,-13.4)`
- `matD = (5.5,-8.2)`

Then use matrix addition and subtraction to compute and display the values of the following matrices:

- `matE = matA + matB`
- `matF = matC - matD`

Finally, test `matE` and `matF` for equality and display the result, replacing the question mark with either `true` or `false`.

Cause the program to display your name in some manner.

My version of the program produces the text output shown in [Figure 4](#), and you should use that as a display template.

Figure 4 Text output from Exercise 1.

```
Prof. Baldwin
matE = 3.0, -5.2
matF = 3.0, -5.2000000000000001
matE equals matF: ?
```

Exercise 2

Using Java and the game-math library named **GM2D03**, or using a different programming environment of your choice, write a program that creates and displays (*in text form*) three vectors having the names and values shown in [Figure 5](#).

Cause your program to test **vecA** for equality, first against **vecB**, and then against **vecC**, and display the results of the tests in place of the question marks shown in [Figure 5](#).

Figure 5 . Text output from Exercise 2.

```
Prof. Baldwin  
vecA = -1.5,2.6  
vecB = -1.5,2.6  
vecC = 8.5,-13.4  
vecA equals vecB: ?  
vecA equals vecC: ?
```

Exercise 3

Using Java and the game-math library named **GM2D03** , or using a different programming environment of your choice, write a program that creates and displays four column matrices having the names and values shown in [Figure 6](#).

Use the first two matrices to create a displacement vector named **dispVecE** and display its value in place of the question marks in [Figure 6](#).

Use the last two matrices to create another displacement vector named **dispVecF** and display its value in place of the question marks in [Figure 6](#).

Test **dispVecE** and **dispVecF** for equality and display the result of the test in place of the final question mark in [Figure 6](#).

Figure 6 . Text output from Exercise 3.

Figure 6 . Text output from Exercise 3.

```
Prof. Baldwin  
matA = 1.5,-2.6  
matB = -1.5,2.6  
dispVecE = ?,?  
matC = 8.5,-13.4  
matD = 5.5,-8.2  
dispVecF = ?,?  
dispVecE equals dispVecF: ?
```

-end-

Game0115r Review

This module contains review questions and answers keyed to the module titled [GAME 2302-0115: Working with Column Matrices, Points, and Vectors](#).

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module contains review questions and answers keyed to the module titled [GAME 2302-0115: Working with Column Matrices, Points, and Vectors](#).

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

A method named **equals** is inherited from the **Object** class and overridden in several classes in the **GM2D03** game-math library.

True or False?

[Answer 1](#)

Question 2

The overridden **equals** method in the **GM2D03.Point** class returns a value of type **double** .

True or False?

[Answer 2](#)

Question 3

It is not difficult to compare **double** and **float** values for absolute equality.

True or False?

[Answer 3](#)

Question 4

The overridden equals method of the **GM2D03.Point** class compares two objects of type **GM2D03.ColMatrix** for equality.

True or False?

[Answer 4](#)

Question 5

The difference between two points is a displacement vector.

True or False?

[Answer 5](#)

Question 6

The **getDisplacementVector** method of the **GM2D03.Point** class computes the sums of the x and y components of the points that are stored in two column matrix objects.

True or False?

[Answer 6](#)

Question 7

The **getDisplacementVector** method of the **GM2D03.Point** class returns a value of type **GM2D03.ColMatrix** .

True or False?

[Answer 7](#)

Question 8

The displacement vector that is returned by the **getDisplacementVector** method of the **GM2D03.Point** class points from the **Point** object on which the method is called to the **Point** object passed as a parameter to the method.

True or False?

[Answer 8](#)

Question 9

Kjell describes the component parts of the a displacement vector

- as the distance you would have to walk,
- first along the x-axis and
- then along the y-axis
- to get from the first point to the second point.

True or False?

[Answer 9](#)

Question 10

The add method of the **GM2D03.ColMatrix** class adds one **ColMatrix** object to another **ColMatrix** object, returning a **ColMatrix** object.

True or False?

[Answer 10](#)

Question 11

The order in which you add column matrix objects has a significant impact on the result.

True or False?

[Answer 11](#)

Question 12

The order in which two column matrix objects are subtracted doesn't matter. The result is the same either way.

True or False?

Answer 12

What is the meaning of the following two images?

This image was inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 12

False. As you should have learned from your studies of the Kjell tutorial, the order in which the two column matrix objects are subtracted has a significant impact on the result.

[Back to Question 12](#)

Answer 11

False. As you should have learned from your studies of the Kjell tutorial, the order in which the two column matrix objects are added doesn't matter. The result is the same either way.

[Back to Question 11](#)

Answer 10

True

[Back to Question 10](#)

Answer 9

True

[Back to Question 9](#)

Answer 8

True

[Back to Question 8](#)

Answer 7

False. The **getDisplacementVector** method of the **GM2D03.Point** class returns a value of type *GM2D03.Vector* .

[Back to Question 7](#)

Answer 6

False. The **getDisplacementVector** method of the **GM2D03.Point** class computes the *differences* (*subtraction*) of the x and y components of the points that are stored in two column matrix objects.

[Back to Question 6](#)

Answer 5

True

[Back to Question 5](#)

Answer 4

True. In effect, this method actually compares two objects of the **ColMatrix** class by calling the equals method defined in the **GM2D03.ColMatrix** class. This is because the values that define a point are actually stored in a column matrix.

[Back to Question 4](#)

Answer 3

False. There is an inherent problem when comparing two **double** or **float** values for equality. If you perform a series of computations twice, using a different computational order for each set of computations, you are likely to end up with two values that are not absolutely equal even if they should be equal. Arithmetic inaccuracies along the way may cause the two results to differ ever so slightly. However, they may be equal from a practical viewpoint.

[Back to Question 3](#)

Answer 2

False. The overridden **equals** method in the GM2D03.Point class returns a value of type **boolean** .

[Back to Question 2](#)

Answer 1

True

[Back to Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0115r Review: Working with Column Matrices, Points, and Vectors
- File: Game0115r.htm
- Published: 12/31/12
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0120: Visualizing Column Matrices

Learn how to display column matrices in a graphical format to make it easier to visualize them.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
- [Homework assignment](#)
- [Run the program](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listing](#)
- [Exercises](#)
 - [Exercise 1](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, you learned:

- How to compare column matrices for equality
- How to compare two points for equality

- How to compare two vectors for equality
- How to add one column matrix to another
- How to subtract one column matrix from another
- How to get a displacement vector from one point to another.

What you will learn

In this module, you will learn how to display column matrices in a graphical format. This may help you to get a better grasp of the nature of column matrices and the results of adding, subtracting, and comparing them.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Sample graphical program output.
- [Figure 2](#). Text output from the program.
- [Figure 3](#). Graphical output for equal vectors.
- [Figure 4](#). Text output for equal vectors.
- [Figure 5](#). Graphic output from Exercise 01.

Listings

- [Listing 1](#). Beginning of the method named displayColumnMatrices.
- [Listing 2](#). Use the slider values to create the two matrices.
- [Listing 3](#). Add and subtract the matrices.
- [Listing 4](#). Display text information about the matrices.
- [Listing 5](#). Create mathematical points.

- [Listing 6](#). Create mathematical displacement vectors.
- [Listing 7](#). Produce a graphical representation of the displacement vectors.
- [Listing 8](#). Source code for the program named ColMatrixVis01.

Preview

Abstract mathematical concepts are often easier to grasp if you can visualize them in graphical format. For example, the nature of the following equation often becomes more apparent once you learn that it is the equation of a straight line.

$$y = m \cdot x + b$$

Similarly, the nature of the following equation often becomes more apparent once you learn that it is the equation of a parabola and you learn the general shape of a parabola.

$$y = x^2 + k$$

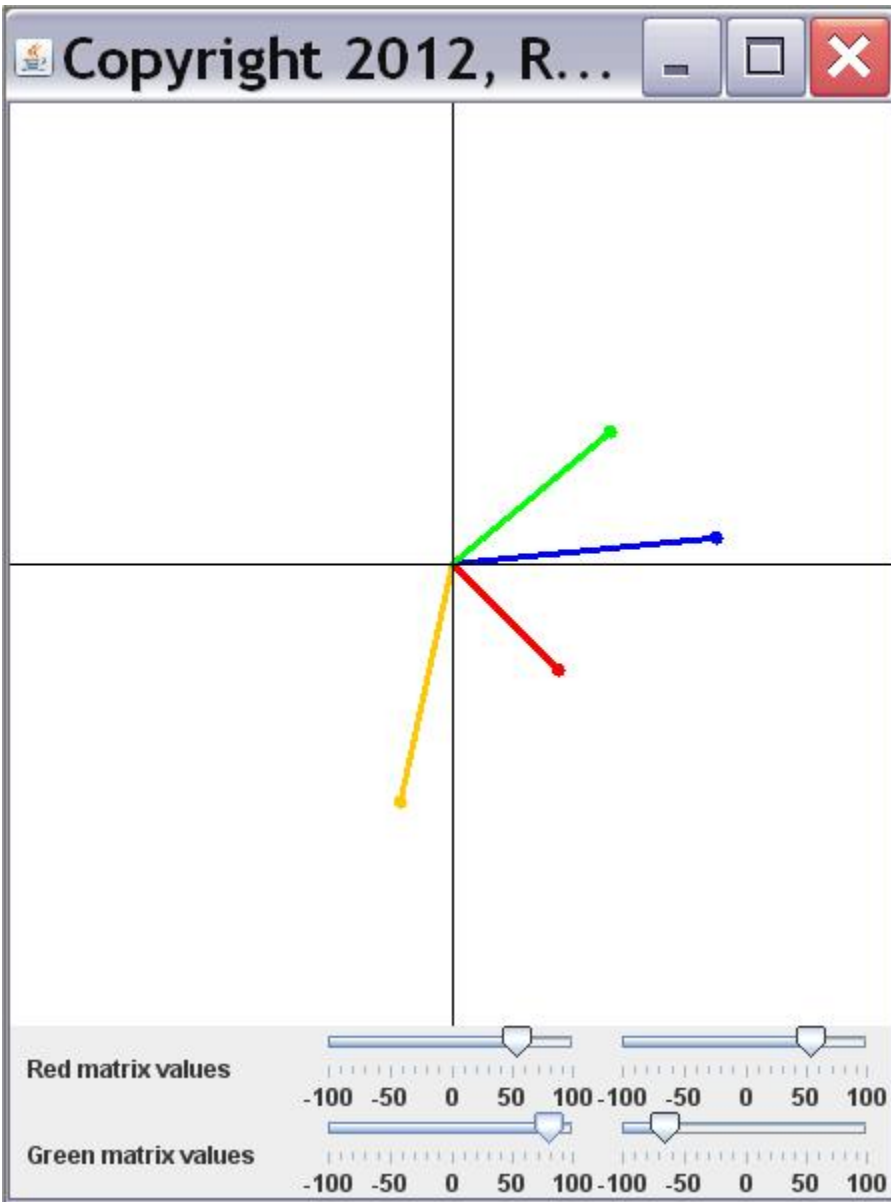
where x^2 indicates x raised to the second power.

As mentioned earlier, in this module, you will learn how to display column matrices in a graphical format. This may help you to get a better grasp of the nature of column matrices and the results of adding, subtracting, and comparing them.

I will present and explain an interactive program that behaves as follows:

Two column matrices are created using values obtained from the sliders shown at the bottom of [Figure 1](#). One matrix is named **redMatrix** and the other matrix is named **greenMatrix**.

Figure 1 Sample graphical program output.



Two additional column matrices are created by adding and subtracting the original matrices. The matrix created by adding the red and green matrices is named **blueMatrix** . The matrix created by subtracting the green matrix from the red matrix is named **orangeMatrix** .

Mathematical points are created to represent the values in the four matrices in a 2D reference frame. Then, mathematical displacement vectors are created for each of the points relative to the origin.

Graphical objects are created for each of the four displacement vectors and those objects are drawn along with Cartesian coordinate axes in the 2D reference frame.

The vectors are shown in the top portion of [Figure 1](#). The red and green vectors represent the red and green matrices. The blue and orange vectors represent the sum and the difference of the red and green matrices.

Text output is displayed to show the matrix values as well as whether the two original matrices are equal. The text values corresponding to the vectors in [Figure 1](#) are shown in [Figure 2](#).

Figure 2 . Text output from the program.

```
redMatrix = 53.0,53.0
greenMatrix = 79.0,-66.0
redMatrix equals greenMatrix: false
blueMatrix = redMatrix + greenMatrix =
132.0,-13.0
orangeMatrix = redMatrix - greenMatrix =
-26.0,119.0
```

If you carefully adjust the sliders so that the two values contained in the **redMatrix** are the same as the two values contained in the **greenMatrix**, the red and green vectors will overlay as shown in [Figure 3](#) and the third line of output text will show true as shown in [Figure 4](#). In this case, only the green vector and part of the blue (*sum*) vector are visible. The red vector is covered by the green vector, and the orange (*difference*) vector has a zero length.

Figure 3 Graphical output for equal vectors.

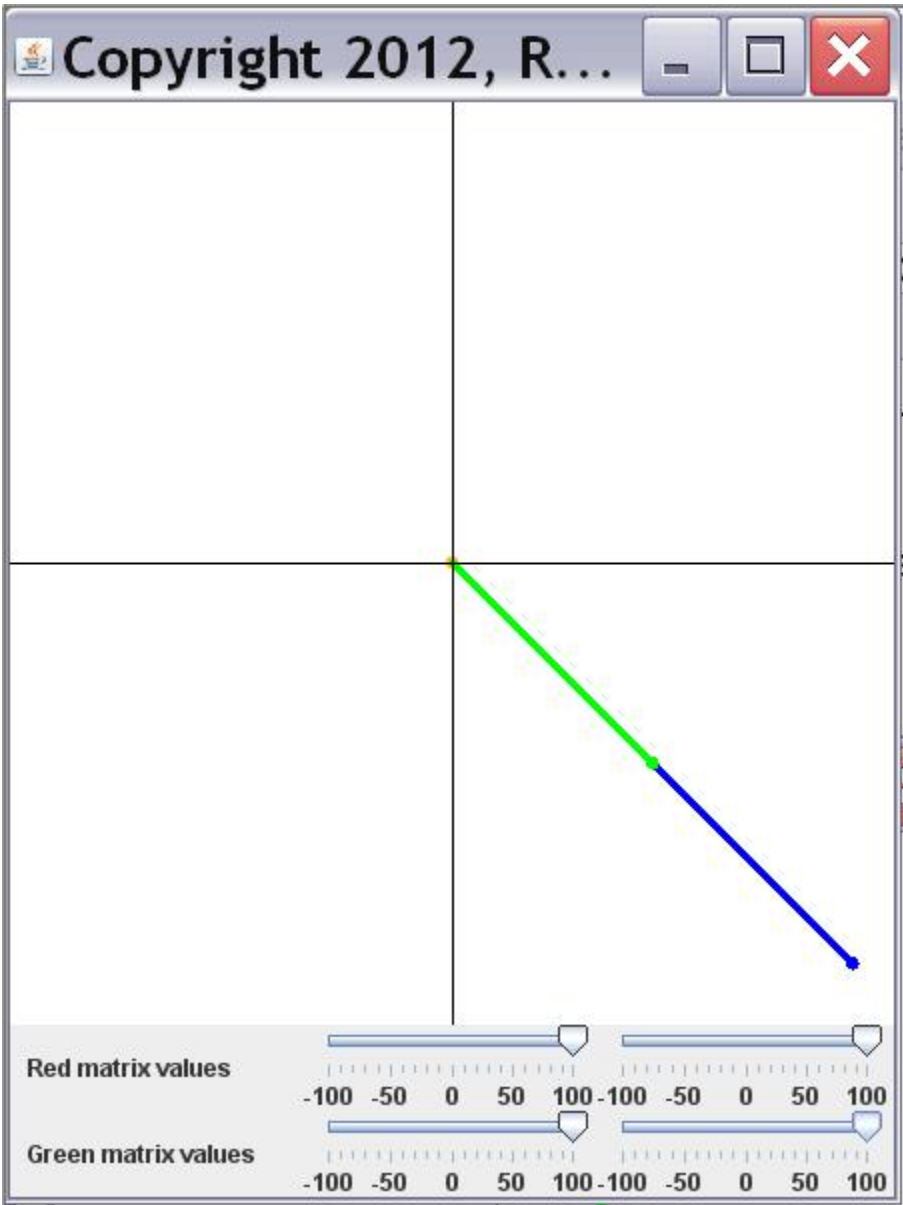


Figure 4 . Text output for equal vectors.

Figure 4 . Text output for equal vectors.

```
redMatrix = 100.0,100.0
greenMatrix = 100.0,100.0
redMatrix equals greenMatrix: true
blueMatrix = redMatrix + greenMatrix =
200.0,200.0
orangeMatrix = redMatrix - greenMatrix =
0.0,0.0
```

There are many other interesting combinations that I could show. However, I will leave it as an exercise for the student to copy, compile, and run the program and use the sliders to experiment with different matrix values.

I will also provide an exercise for you to complete on your own at the end of the module. The exercise will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

Because of its interactive nature, much of the code in this program is at a complexity level that is beyond the scope of this course. However, most of the interesting work is done in the method named **displayColumnMatrices** and I will concentrate on explaining that method.

You can view a complete listing of the program named **ColMatrixVis01** in [Listing 8](#) near the end of the module.

Note that the program requires access to the game library named **GM2D03** . The source code for that library was provided in the earlier module titled [GAME2302-0115: Working with Column Matrices, Points, and Vectors](#) and you can copy it from there.

The method named **displayColumnMatrices**

The purpose of this method is to:

1. Create two column matrices named **redMatrix** and **greenMatrix** using values obtained from sliders.
2. Create two more column matrices named **blueMatrix** and **orangeMatrix** by adding and subtracting the red and green matrices.
3. Display text information about the matrices including whether or not the red and green matrices are equal.
4. Create mathematical points in a 2D coordinate frame that represent the values in the matrices.
5. Create mathematical displacement vectors that represent the displacements of each of the points relative to the origin.
6. Create and draw graphics objects that represent each of the mathematical displacement vectors along with Cartesian coordinate axes for the 2D reference frame.

Beginning of the method named displayColumnMatrices

I will explain the method named **displayColumnMatrices** in fragments. You can view the entire method in [Listing 8](#). The first fragment is shown in [Listing 1](#).

Listing 1 . Beginning of the method named displayColumnMatrices.

Listing 1 . Beginning of the method named displayColumnMatrices.

```
void displayColumnMatrices(Graphics2D g2D){  
  
    //Get two values for each matrix from the  
    sliders.  
    red0 = redSlider0.getValue();  
    red1 = redSlider1.getValue();  
    green0 = greenSlider0.getValue();  
    green1 = greenSlider1.getValue();  
}
```

Get two values for each matrix from the sliders

The little things with the pointed bottoms on the sliders in [Figure 1](#) are often called the *thumbs* of the sliders. Each thumb points down to a numeric scale that ranges from -100 on the left to +100 on the right.

Each time you move a thumb on a slider, the method named **displayColumnMatrices** , including the code in [Listing 1](#) , is executed. The code in [Listing 1](#) gets the value corresponding to the position of each thumb and saves those four values in the variables named **red0** , **red1** , **green0** , and **green1** .

The two sliders in the top row represent red. The two sliders in the bottom row represent green.

The values of the two sliders on the left correspond to **red0** and **green0** . The two on the right correspond to **red1** and **green1** .

Use the slider values to create the two matrices

[Listing 2](#) uses the slider values to create the two matrices named **redMatrix** and **greenMatrix** .

*(More properly, the code uses the values to create two **ColMatrix** objects and to store references to those objects in the variables named **redMatrix** and **greenMatrix** .)*

Listing 2 . Use the slider values to create the two matrices.

```
//Use the slider values to create the two
matrices
// named redMatrix and greenMatrix.
GM2D03.ColMatrix redMatrix =
                        new
GM2D03.ColMatrix(red0,red1);
GM2D03.ColMatrix greenMatrix =
                        new
GM2D03.ColMatrix(green0,green1);
```

There is nothing new in [Listing 2](#) that you haven't seen before so further explanation should not be necessary.

Add and subtract the matrices

[Listing 3](#) creates two additional matrices by adding and subtracting the red and green matrices. References to the new matrices are stored in the variables named **blueMatrix** and **orangeMatrix** .

Listing 3 . Add and subtract the matrices.

```
        //Create two additional matrices by adding
and
        // subtracting the red and green matrices.
        GM2D03.ColMatrix blueMatrix =

        redMatrix.add(greenMatrix);
        GM2D03.ColMatrix orangeMatrix =

        redMatrix.subtract(greenMatrix);
```

Once again, there is nothing new in [Listing 3](#), so further explanation should not be necessary.

Display text information about the matrices

[Listing 4](#) displays text information about the four matrices, including whether or not the red and green matrices are equal.

Listing 4 . Display text information about the matrices.

Listing 4 . Display text information about the matrices.

```
//Display text information about the
matrices.
System.out.println();//blank line
System.out.println("redMatrix = " +
redMatrix);
System.out.println("greenMatrix = " +
greenMatrix);
System.out.println("redMatrix equals
greenMatrix: " +

redMatrix.equals(greenMatrix));
System.out.println(
    "blueMatrix = redMatrix +
greenMatrix = " +

blueMatrix);
System.out.println(
    "orangeMatrix = redMatrix -
greenMatrix = " +

orangeMatrix);
```

The code in [Listing 4](#) produced the text in [Figure 2](#) and [Figure 4](#).

Displaying information about the matrices

There are many ways to display information about matrices, including the simple text displays shown in [Figure 2](#) and [Figure 4](#). The problem with text displays is that you have to study the numbers in detail to get a feel for an individual matrix and a feel for the relationships among two or more matrices.

A graphical display can often convey that sort of information at first glance. Then you are faced with a decision as to how you should construct the graphical display.

For the case of a column matrix with two elements, a good approach is to let the two matrix values represent the x and y coordinate values of a mathematical point in a 2D reference frame and then to display information about the point. That is the approach taken by this program.

Create mathematical points

[Listing 5](#) creates mathematical points in a 2D coordinate frame that represent the values in the matrices. [Listing 5](#) also creates a point that represents the origin.

Listing 5 . Create mathematical points.

Listing 5 . Create mathematical points.

```
//Create mathematical points in a 2D
coordinate
// frame that represent the values in the
matrices.
// Also create a point that represents the
origin.
GM2D03.Point origin =
    new GM2D03.Point(new
GM2D03.ColMatrix(0,0));
GM2D03.Point redPoint =
    new
GM2D03.Point(redMatrix);
GM2D03.Point greenPoint =
    new
GM2D03.Point(greenMatrix);
GM2D03.Point bluePoint = new
GM2D03.Point(blueMatrix);
GM2D03.Point orangePoint =
    new
GM2D03.Point(orangeMatrix);
```

Displaying the points

Once you have the points, you then need to decide what might be the best format in which to display them. One obvious approach would simply be to draw small symbols in the 2D coordinate frame that represent the locations of the points.

However, in most real-world situations, we tend to evaluate the value of something relative to a value of zero.

(There are, however, exceptions to this rule. For example, when considering the temperature in Celsius, we tend to evaluate the temperature relative to zero degrees Celsius, which is the freezing point of water. On a Fahrenheit scale, however, we tend to evaluate temperature relative to 32-degrees F, which is the freezing point of water.)

Displacement vectors

A good way to get a feel for the location of a mathematical point in a 2D reference frame is to compare the location of that point with the location of a different point through the use of a displacement vector. That is the approach taken by this program with the anchor point being the point at the origin against which all other points are compared.

[Listing 6](#) creates mathematical displacement vectors that represent the displacements of each of the four points created earlier relative to the origin.

Listing 6 . Create mathematical displacement vectors.

Listing 6 . Create mathematical displacement vectors.

```
//Create mathematical displacement vectors
that
// represent the displacements of each of
the points
// relative to the origin.
GM2D03.Vector redVec =

origin.getDisplacementVector(redPoint);
GM2D03.Vector greenVec =

origin.getDisplacementVector(greenPoint);
GM2D03.Vector blueVec =

origin.getDisplacementVector(bluePoint);
GM2D03.Vector orangeVec =

origin.getDisplacementVector(orangePoint);
```

Produce a graphical representation of the displacement vectors

[Listing 7](#) produces and displays a graphical representation of each of the four displacement vectors as shown in [Figure 1](#).

Listing 7 . Produce a graphical representation of the displacement vectors.

Listing 7 . Produce a graphical representation of the displacement vectors.

```
//The remaining code is used to create and
draw
// graphical objects.
//Erase the off-screen image
g2D.setColor(Color.WHITE);
g2D.fill(rect);

//Set the line thickness so that the
vectors will be
// drawn with a heavy line.
g2D.setStroke(new BasicStroke(3));

//Draw the four vectors with their tails
at the
// origin.
g2D.setColor(Color.BLUE);
blueVec.draw(g2D,origin);

g2D.setColor(Color.ORANGE);
orangeVec.draw(g2D,origin);

g2D.setColor(Color.RED);
redVec.draw(g2D,origin);

g2D.setColor(Color.GREEN);
greenVec.draw(g2D,origin);

//Draw the axes with thinner lines.
g2D.setStroke(new BasicStroke(1));
g2D.setColor(Color.BLACK);
drawAxes(g2D);

} //end displayColumnMatrices
```

Listing 7 . Produce a graphical representation of the displacement vectors.

There is very little new code in [Listing 7](#), and the new code that is there should be easy to understand.

Analysis of results

The head of the red vector in [Figure 1](#) represents the two values in the column matrix known as **redMatrix** . The length and direction of that vector shows how it relates to a column vector having two elements, each with a value of zero.

Similarly, the head of the green vector in [Figure 1](#) represents the two values in the column matrix known as **greenMatrix** .

The head of the blue vector represents the two values in the column matrix known as **blueMatrix** , which was created by adding the red and green matrices. In case you haven't noticed, a line drawn from the head of the red vector to the head of the blue vector would have the same length and direction as the green vector. This will come up again in a future module when we discuss the vector addition parallelogram.

The head of the orange vector represents the two values in the column matrix known as **orangeMatrix** , which was created by subtracting the green matrix from the red matrix. Again, a line drawn from the head of the red vector to the head of the orange vector would have the same length and opposite direction as the green vector.

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through Chapter 3 - *Vector Addition* . That is also the homework assignment for the next module.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the program

I encourage you to copy the code from [Listing 8](#). Compile the code and execute it in conjunction with the game-math library named **GM2D03**. The source code for that library was provided in the earlier module titled [GAME2302-0115: Working with Column Matrices, Points, and Vectors](#) and you can copy it from there.. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to create column matrices using values obtained from sliders. You learned how to create additional column matrices by adding and subtracting the original matrices.

You learned how to display text information about the matrices including whether or not they are equal.

You learned how to display the matrices in a graphical format where each matrix is represented by a displacement vector in a 2D reference frame.

What's next?

In the next module, you will learn:

- How to add two or more vectors.
- About the head-to-tail rule in vector addition.
- About the vector addition parallelogram.
- About the relationship between the length of the sum of vectors and the sum of the lengths of vectors.

- How to add a vector to a point.
- How to get the length of a vector.
- How to represent an object in different coordinate frames.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME2302-0120: Visualizing Column Matrices
- File: Game0120.htm
- Published: 10/15/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

A complete listing of the program named **ColMatrixVis01** is provided in [Listing 8](#). The game library named **GM2D03**, which is required for compiling and executing this program, was provided in the earlier module titled [GAME2302-0115: Working with Column Matrices, Points, and Vectors](#). You can copy it from there.

Listing 8 . Source code for the program named ColMatrixVis01.

```
/*ColMatrixVis01.java  
Copyright 2012, R.G.Baldwin
```

The purpose of this program is to help the student visualize column matrices. Two column matrices are created using values obtained from sliders.

Two additional column matrices are created by adding and subtracting the original matrices.

Mathematical points are created to represent the values in the matrices in a 2D reference frame.

Displacement vectors are created for each of the points relative to the origin.

The vectors are drawn along with Cartesian coordinate axes in the 2D reference frame.

Text output is displayed to show the matrix values as well as whether the two original matrices are equal.

Tested using JDK 1.7 under WinXP and Windows 7

*****/

```
import java.awt.*;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.JLabel;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import java.lang.Math;
import java.util.*;
```

```
class ColMatrixVis01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class ColMatrixVis01
//=====
=====//
```

```
class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 450;

    int vSize = 600;
    Image osi;//off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    //Panel to contain the sliders and associated
    labels.
    private JPanel mainPanel = new JPanel();
```

```
//Sliders used to produce values for column  
matrices.
```

```
private JSlider redSlider0 = new JSlider();  
private JSlider redSlider1 = new JSlider();  
private JSlider greenSlider0 = new JSlider();  
private JSlider greenSlider1 = new JSlider();
```

```
//Storage area for values extracted from  
sliders.
```

```
private int red0 = 100;  
private int red1 = 100;  
private int green0 = 100;  
private int green1 = -100;
```

```
//Object used to erase the off-screen image.  
private Rectangle rect;
```

```
GUI(){//constructor
```

```
    //Configure the sliders.  
    redSlider0.setMaximum(100);  
    redSlider0.setMinimum(-100);  
    redSlider0.setMajorTickSpacing(50);  
    redSlider0.setMinorTickSpacing(10);  
    redSlider0.setPaintTicks(true);  
    redSlider0.setPaintLabels(true);  
    redSlider0.setValue(red0);
```

```
    redSlider1.setMaximum(100);  
    redSlider1.setMinimum(-100);  
    redSlider1.setMajorTickSpacing(50);  
    redSlider1.setMinorTickSpacing(10);  
    redSlider1.setPaintTicks(true);  
    redSlider1.setPaintLabels(true);  
    redSlider1.setValue(red1);
```

```
    greenSlider0.setMaximum(100);  
    greenSlider0.setMinimum(-100);
```

```

greenSlider0.setMajorTickSpacing(50);
greenSlider0.setMinorTickSpacing(10);
greenSlider0.setPaintTicks(true);
greenSlider0.setPaintLabels(true);
greenSlider0.setValue(green0);

greenSlider1.setMaximum(100);
greenSlider1.setMinimum(-100);
greenSlider1.setMajorTickSpacing(50);
greenSlider1.setMinorTickSpacing(10);
greenSlider1.setPaintTicks(true);
greenSlider1.setPaintLabels(true);
greenSlider1.setValue(green1);

//Set the layout manager for the panel that
contains
// the sliders and the associated labels.
mainPanel.setLayout(new GridLayout(2,3));

//Add the sliders and associated labels to the
panel.
mainPanel.add(new JLabel("    Red matrix
values"));
mainPanel.add(redSlider0);
mainPanel.add(redSlider1);

mainPanel.add(new JLabel("    Green matrix
values"));
mainPanel.add(greenSlider0);
mainPanel.add(greenSlider1);

//Set JFrame size, title, and close operation.
setSize(hSize,vSize);
setTitle("Copyright 2012, R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```

    //Create a new drawing canvas and add it to
the
    // center of the JFrame.
    myCanvas = new MyCanvas();
    this.getContentPane().add(myCanvas);
    this.getContentPane().add(

mainPanel, BorderLayout.SOUTH);

    //This object must be visible before you can
get an
    // off-screen image. It must also be visible
before
    // you can compute the size of the canvas.
    setVisible(true);
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

    //Configure the object that will be used to
erase
    // the off-screen image.
    rect = new Rectangle(
        -osiWidth/2, -
osiHeight/2, osiWidth, osiHeight);

    //Create an off-screen image and get a
graphics
    // context on it.
    osi = createImage(osiWidth, osiHeight);
    final Graphics2D g2D =
        (Graphics2D)
(osi.getGraphics());

    //Translate the origin to the center of the
// off-screen image.
    g2D.translate(osiWidth/2.0, osiHeight/2.0);

```

```

//Erase the off-screen image.
g2D.setColor(Color.WHITE);
g2D.fill(rect);//erase the osi

//Display the initial values of the column
matrices
displayColumnMatrices(g2D);

//Register a listener on each of the sliders.
redSlider0.addChangeListener(
    new ChangeListener(){
        public void stateChanged(ChangeEvent e){
            //Re-display the column matrices each
time the
            // thumb is moved on the slider.
            displayColumnMatrices(g2D);
        }//end stateChanged
    }//end new ChangeListener
);//end addChangeListener

redSlider1.addChangeListener(
    new ChangeListener(){
        public void stateChanged(ChangeEvent e){
            displayColumnMatrices(g2D);
        }//end stateChanged
    }//end new ChangeListener
);//end addChangeListener

greenSlider0.addChangeListener(
    new ChangeListener(){
        public void stateChanged(ChangeEvent e){
            displayColumnMatrices(g2D);
        }//end stateChanged
    }//end new ChangeListener
);//end addChangeListener

greenSlider1.addChangeListener(

```

```

        new ChangeListener(){
            public void stateChanged(ChangeEvent e){
                displayColumnMatrices(g2D);
            }//end stateChanged
        }//end new ChangeListener
    );//end addChangeListener
} //end constructor
//-----
-----//

//The purpose of this method is to
// 1. Create two column matrices named redMatrix
and
//    greenMatrix using values obtained from
sliders.
// 2. Create two more column matrices named
blueMatrix
//    and orangeMatrix by adding and subtracting
the red
//    and green matrices.
// 3. Display text information about the
matrices
//    including whether the red and green
matrices are
//    equal.
// 4. Create mathematical points in a 2D
coordinate

//    frame that represents the values in the
matrices.
// 5. Create mathematical displacement vectors
that
//    represent the displacements of each of the
points
//    relative to the origin.
// 6. Create and draw graphics objects that
represent
//    each of the mathematical displacement

```

```

vectors
    //      along with Cartesian coordinate axes for
the
    //      2D reference frame.
    void displayColumnMatrices(Graphics2D g2D){

        //Get two values for each matrix from the
sliders.
        red0 = redSlider0.getValue();
        red1 = redSlider1.getValue();
        green0 = greenSlider0.getValue();
        green1 = greenSlider1.getValue();

        //Use the slider values to create the two
matrices
        // named redMatrix and greenMatrix.
        GM2D03.ColMatrix redMatrix =
                                new
GM2D03.ColMatrix(red0,red1);
        GM2D03.ColMatrix greenMatrix =
                                new
GM2D03.ColMatrix(green0,green1);

        //Create two additional matrices by adding and
// subtracting the red and green matrices.
        GM2D03.ColMatrix blueMatrix =

redMatrix.add(greenMatrix);
        GM2D03.ColMatrix orangeMatrix =

redMatrix.subtract(greenMatrix);

        //Display text information about the matrices.
        System.out.println();//blank line
        System.out.println("redMatrix = " +
redMatrix);
        System.out.println("greenMatrix = " +

```



```

greenMatrix);
    System.out.println("redMatrix equals
greenMatrix: " +

redMatrix.equals(greenMatrix));
    System.out.println(
        "blueMatrix = redMatrix +
greenMatrix = " +

blueMatrix);
    System.out.println(
        "orangeMatrix = redMatrix -
greenMatrix = " +

orangeMatrix);

    //Create mathematical points in a 2D
coordinate
    // frame that represent the values in the
matrices.
    // Also create a point that represents the
origin.
    GM2D03.Point origin =
        new GM2D03.Point(new
GM2D03.ColMatrix(0,0));
    GM2D03.Point redPoint =
        new
GM2D03.Point(redMatrix);
    GM2D03.Point greenPoint =
        new
GM2D03.Point(greenMatrix);
    GM2D03.Point bluePoint = new
GM2D03.Point(blueMatrix);
    GM2D03.Point orangePoint =
        new
GM2D03.Point(orangeMatrix);

```

```

        //Create mathematical displacement vectors
that
        // represent the displacements of each of the
points
        // relative to the origin.
        GM2D03.Vector redVec =

origin.getDisplacementVector(redPoint);
        GM2D03.Vector greenVec =

origin.getDisplacementVector(greenPoint);
        GM2D03.Vector blueVec =

origin.getDisplacementVector(bluePoint);
        GM2D03.Vector orangeVec =

origin.getDisplacementVector(orangePoint);

        //The remaining code is used to create and
draw
        // graphical objects.
        //Erase the off-screen image
        g2D.setColor(Color.WHITE);
        g2D.fill(rect);

        //Set the line thickness so that the vectors
will be
        // drawn with a heavy line.
        g2D.setStroke(new BasicStroke(3));

        //Draw the four vectors with their tails at
the
        // origin.
        g2D.setColor(Color.BLUE);
        blueVec.draw(g2D,origin);

        g2D.setColor(Color.ORANGE);

```

```

    orangeVec.draw(g2D, origin);

    g2D.setColor(Color.RED);
    redVec.draw(g2D, origin);

    g2D.setColor(Color.GREEN);
    greenVec.draw(g2D, origin);

    //Draw the axes with thinner lines.
    g2D.setStroke(new BasicStroke(1));
    g2D.setColor(Color.BLACK);
    drawAxes(g2D);

} //end displayColumnMatrices
//-----
-----//

//The purpose of this method is to draw a pair
of
// Cartesian coordinate axes onto the
// off-screen image.
void drawAxes(Graphics2D g2D){

    //Define four points at the edges of the
coordinate
    // frame and the ends of the axes.
    GM2D03.Point point0 = new GM2D03.Point(
                                new GM2D03.ColMatrix(-
osiWidth/2, 0));
    GM2D03.Point point1 = new GM2D03.Point(
                                new
GM2D03.ColMatrix(osiWidth/2, 0));
    GM2D03.Point point2 = new GM2D03.Point(
                                new GM2D03.ColMatrix(0, -
osiHeight/2));
    GM2D03.Point point3 = new GM2D03.Point(
                                new

```

```

GM2D03.ColMatrix(0,osiHeight/2));

    //Now define the two lines based on the end
points..
    GM2D03.Line xAxis = new
GM2D03.Line(point0,point1);
    GM2D03.Line yAxis = new
GM2D03.Line(point2,point3);

    //Now draw a visual manifestation of each line
    // on g2D.
    xAxis.draw(g2D);
    yAxis.draw(g2D);

    //Repaint the display area
    myCanvas.repaint();

} //end drawAxes

//=====
====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the
        // screen or when the repaint method is called
on the
        // Canvas object.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        } //end overridden paint()

    } //end inner class MyCanvas

```

```

} //end class GUI
//=====
=====//

```

Exercises

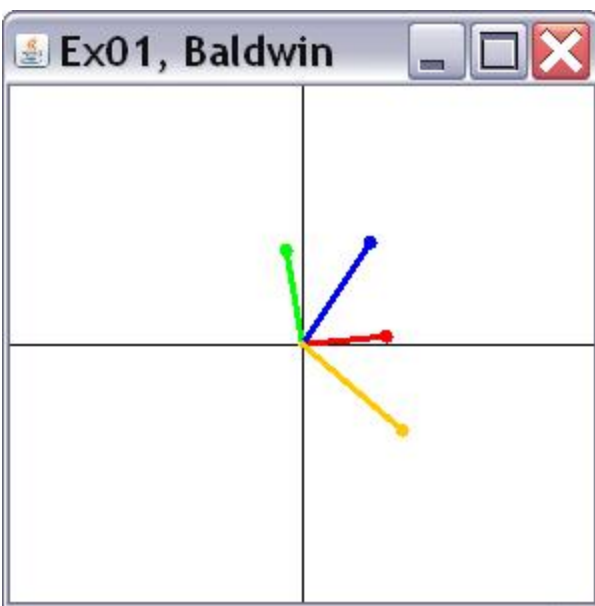
Exercise 1

Using Java and the game-math library named **GM2D03** , or using a different programming environment of your choice, write a program that uses random values to generate two column matrix objects.

Generate two more column matrix objects as the sum and difference of the two original column matrix objects.

Display the two original column matrix objects in red and green and display the sum and difference matrix objects in blue and orange as shown in [Figure 5](#).

Figure 5 . Graphic output from Exercise 01.



-end-

Game0120r Review

This module contains review questions and answers keyed to the module titled GAME 2302-0120: Visualizing Column Matrices.

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#)
- [Figures](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module contains review questions and answers keyed to the module titled [GAME 2302-0120: Visualizing Column Matrices](#).

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: The following equation is the equation of a straight line in two dimensions where m is the slope, b is the y-intercept, and the asterisk indicates multiplication:

$$y = m * x + b$$

[Answer 1](#)

Question 2

True or False: The following equation is the equation of a hyperbola in two dimensions where x^2 indicates x raised to the second power and k is the y -intercept:

$$y = x^2 + k$$

[Answer 2](#)

Question 3

In this and other questions that follow, the upper-case T represents transpose, as shown by a superscript T in the Kjell tutorial.

True or False: Given that a and b are column matrices, the following equations are true:

$$a = (3,2)^T$$

$$b = (-2,1)^T$$

$$a + b = (5,3)^T$$

[Answer 3](#)

Question 4

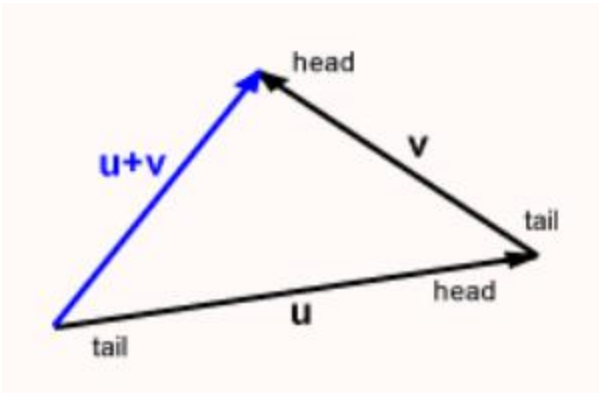
True or False: The addition of vectors depends on their location.

[Answer 4](#)

Question 5

Consider [Figure 1](#).

Figure 1 Question 5



True or False: A valid statement of the head-to-tail rule is:

Head-to-Tail Rule: Move vector v (keeping its length and orientation the same) until its tail touches the tail of u . The sum is the vector from the tail of u to the head of v .

[Answer 5](#)

Question 6

True or False: The head-to-tail rule works in both 2D and 3D.

[Answer 6](#)

Question 7

True or False: The associative rule applies to the addition of vectors in all dimensions.

[Answer 7](#)

Question 8

True or False: Vector addition is commutative, just like addition of real numbers.

[Answer 8](#)

Question 9

True or False: The commutative rule for vectors applies in all dimensions.

[Answer 9](#)

Question 10

True or False: A coordinate frame consists of a distinguished point P_0 (called the origin) and an axis for each dimension (often called X and Y). In 2D space there are two axes; in 3D space there are three axes (often called X, Y, and Z).

[Answer 10](#)

Question 11

True or False: Points can be represented with column matrices independently of the coordinate frame.

[Answer 11](#)

Question 12

True or False: Because of the triangle inequality, the sum of two vectors is not necessarily the same as the sum of the lengths of the two vectors.

[Answer 12](#)

Question 13

True or False: A displacement vector added to a point results in a point.

[Answer 13](#)

Figures

- [Figure 1](#). Question 5.

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 13

True

Kjell, Chapter 3

[Back to Question 13](#)

Answer 12

True

Kjell, Chapter 3

[Back to Question 12](#)

Answer 11

False. The correct statement is:

Points can be represented with column matrices. To do this, you first need to decide on a coordinate frame (sometimes called just frame).

Kjell, Chapter 3

[Back to Question 11](#)

Answer 10

True

Kjell, Chapter 3

[Back to Question 10](#)

Answer 9

True

Kjell, Chapter 3

[Back to Question 9](#)

Answer 8

True

Kjell, Chapter 3

[Back to Question 8](#)

Answer 7

True

Kjell, Chapter 3

[Back to Question 7](#)

Answer 6

True

Kjell, Chapter 3

[Back to Question 6](#)

Answer 5

False. The correct statement is:

Head-to-Tail Rule: Move vector v (keeping its length and orientation the same) until its tail touches the head of u . The sum is the vector from the tail of u to the head of v .

Kjell, Chapter 3

[Back to Question 5](#)

Answer 4

False. Vectors have no position or location.

Kjell, Chapter 3.

[Back to Question 4](#)

Answer 3

False

$$a + b = (1,3)^T$$

Kjell Chapter 3

[Back to Question 3](#)

Answer 2

False. This is the equation of a parabola, not a hyperbola.

[Back to Question 2](#)

Answer 1

True

[Back to Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0120r Review: Visualizing Column Matrices

- File: Game0120r.htm
- Published: 09/22/13
- Revised: 01/24/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0125: Vector Addition

Learn how to add two or more vectors. Also learn about the head-to-tail rule in vector addition, about the vector addition parallelogram, about the relationship between the length of the sum of vectors and the lengths of the individual vectors in the sum, how to add a vector to a point, how to get the length of a vector, and how to represent an object in different coordinate frames.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The game-math library named GM2D04](#)
 - [The program named VectorAdd01](#)
 - [The program named CoordinateFrame01](#)
 - [The program named VectorAdd02](#)
 - [The program named VectorAdd03](#)
 - [The program named VectorAdd04](#)
- [Documentation for the GM2D04 library](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous couple of modules, you learned:

- How to compare column matrices for equality
- How to compare two points for equality
- How to compare two vectors for equality
- How to add one column matrix to another
- How to subtract one column matrix from another
- How to get a displacement vector from one point to another

What you will learn

In this module you will learn:

- How to add two or more vectors
- About the head-to-tail rule in vector addition
- About the vector addition parallelogram
- About the relationship between the length of the sum of vectors and the lengths of the individual vectors in the sum
- How to add a vector to a point
- How to get the length of a vector
- How to represent an object in different coordinate frames

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output from the program named VectorAdd01.
- [Figure 2](#). Screen output from CoordinateFrame01 at startup.
- [Figure 3](#). Screen output from CoordinateFrame01 after changes to the coordinate frame.
- [Figure 4](#). Screen output from the program named VectorAdd02.
- [Figure 5](#). Graphic screen output from the program named VectorAdd03.
- [Figure 6](#). Command-line output from the program named VectorAdd03.
- [Figure 7](#). Screen output from the program named VectorAdd04.
- [Figure 8](#). Screen output from Exercise 1.
- [Figure 9](#). Screen output from Exercise 2.

Listings

- [Listing 1](#). The add method of the GM2D04.Vector class.
- [Listing 2](#). The getLength method of the GM2D04.Vector class.
- [Listing 3](#). The addVectorToPoint method of the GM2D04 class.
- [Listing 4](#). Beginning of the program named VectorAdd01.
- [Listing 5](#). Beginning of the method named drawOffScreen.
- [Listing 6](#). The method named setCoordinateFrame.
- [Listing 7](#). Adding two vectors.
- [Listing 8](#). Draw vecA in RED with its tail at the origin.
- [Listing 9](#). Draw vecB in GREEN head-to-tail with vecA.
- [Listing 10](#). Draw sumOf2 in MAGENTA with its tail at the origin.
- [Listing 11](#). Extending the example to three vectors.
- [Listing 12](#). The actionPerformed method.
- [Listing 13](#). The setCoordinateFrame method.
- [Listing 14](#). Beginning of the drawOffScreen method.
- [Listing 15](#). Define a point to position the vectors.
- [Listing 16](#). Remaining code in the drawOffScreen method.
- [Listing 17](#). Beginning of the method named drawOffScreen of the program named VectorAdd02.
- [Listing 18](#). Do the same operations in a different order.

- [Listing 19](#). Source code for the game-math library named GM2D04.
- [Listing 20](#). Source code for the program named VectorAdd01.
- [Listing 21](#). Source code for the program named CoordinateFrame01.
- [Listing 22](#). Source code for the program named VectorAdd02.
- [Listing 23](#). Source code for the program named VectorAdd03.
- [Listing 24](#). Source code for the program named VectorAdd04.

Preview

In this module, I will present and explain several updates to the game-math library. In addition, I will present and explain five sample programs that illustrate the use of the new features of the library.

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

The game-math library was updated and the name was changed to **GM2D04** in preparation for this module. Much of the code in the updated library remains unchanged. I explained that code in the previous modules and I won't repeat that explanation in this module. I will concentrate on explaining the modifications that I made to the library in this module.

The game-math library named GM2D04

A complete listing of the library program is provided in [Listing 19](#) near the end of the module.

This update added the following new capabilities to the library:

- **GM2D04.Vector.add** - Adds this **Vector** object to a **Vector** object received as an incoming parameter and returns the sum as a new **Vector** object.

- **GM2D04.Vector.getLength** - Returns the length of a **Vector** as type **double** .
- **GM2D04.Point.addVectorToPoint** - Adds a **Vector** object to a **Point** object returning a new **Point** object.

These three new methods are presented in [Listing 1](#), [Listing 2](#), and [Listing 3](#) below. These methods are so simple that no explanation should be required for you to understand them.

Listing 1 . The add method of the GM2D04.Vector class.

```
//Adds this vector to a vector received as
an incoming
// parameter and returns the sum as a
vector.
public GM2D04.Vector add(GM2D04.Vector
vec){
    return new GM2D04.Vector(new ColMatrix(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1)));
} //end add
```

Listing 2 . The getLength method of the GM2D04.Vector class.

Listing 2 . The getLength method of the GM2D04.Vector class.

```
//Returns the length of a Vector object.  
public double getLength(){  
    return Math.sqrt(  
        getData(0)*getData(0) +  
        getData(1)*getData(1));  
} //end getLength
```

Listing 3 . The addVectorToPoint method of the GM2D04 class.

```
//Adds a Vector to a Point producing a new  
Point.  
public GM2D04.Point addVectorToPoint(  
    GM2D04.Vector vec){  
    return new GM2D04.Point(new  
    GM2D04.ColMatrix(  
        getData(0) +  
        vec.getData(0),  
        getData(1) +  
        vec.getData(1)));  
} //end addVectorToPoint
```

Simple but powerful

Although these new methods are individually very simple, when combined with the other methods in the library, they significantly increase the power of the library. For example, in the next module I will show you that the library in its current form supports translation and animation.

I will illustrate the use of these new methods in the sample programs that follow.

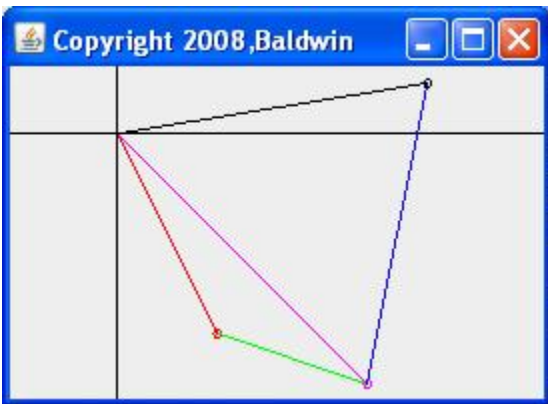
The program named VectorAdd01

This program illustrates the addition of two and then three vectors. It also illustrates the *head-to-tail* rule described by Kjell. A complete listing of the program is provided in [Listing 20](#) near the end of the module.

Screen output from the program named VectorAdd01

The screen output from the program is shown in [Figure 1](#).

Figure 1 Screen output from the program named VectorAdd01.



You will recall that the game-math library represents a **Vector** object graphically as a straight line with a small circle at the head. Thus, there are five vectors drawn in [Figure 1](#). I will explain the meaning of the output in conjunction with the explanation of the program.

Beginning of the program named VectorAdd01

[Listing 4](#) shows the entire class named **VectorAdd01** along with the beginning of the class named **GUI** including the constructor for the **GUI** class.

Listing 4 . Beginning of the program named VectorAdd01.

```
class VectorAdd01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class VectorAdd01
//=====================================================
=====//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 275;
    int vSize = 200;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to
        the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);
```

```

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth,osiHeight);
Graphics2D g2D = (Graphics2D)
(osi.getGraphics());

        //Draw some graphical objects on the off-
screen
        // image that represent underlying data
objects in
        // 2D space.
drawOffScreen(g2D);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.
myCanvas.repaint();

    }//end constructor

```

Very familiar code

The code in [Listing 4](#) is very similar to code that I explained in the earlier module titled [GAME 2302-0110: Updating the Math Library for Graphics](#). Therefore, I won't explain that code again in this module.

This code is mainly needed to get everything set up for graphics.

Note that the code in [Listing 4](#) makes a call to the method named **drawOffScreen** near the end of the listing. That is where we find the interesting code.

Beginning of the method named drawOffScreen

The beginning of the **drawOffScreen** method is shown in [Listing 5](#).

The purpose of this method is to add some **Vector** objects and to cause visual manifestations of the raw **Vector** objects and the resultant **Vector** objects to be drawn onto an off-screen image.

Listing 5 . Beginning of the method named drawOffScreen.

```
void drawOffScreen(Graphics2D g2D){  
    setCoordinateFrame(g2D);
```

[Listing 5](#) begins by calling the method named **setCoordinateFrame** , which is shown in its entirety in [Listing 6](#). I will put the discussion of the **drawOffScreen** method on hold while I explain the method named **setCoordinateFrame** .

The method named setCoordinateFrame

This method sets the origin to a point near the upper-left corner of the off-screen image (see [Figure 1](#)) and draws orthogonal axes on the off-screen image intersecting at the origin.

Listing 6 . The method named setCoordinateFrame.

```
private void setCoordinateFrame(Graphics2D
g2D){
    //Translate the origin.
    g2D.translate(0.2*osiWidth,0.2*osiHeight);

    //Draw new X and Y-axes in default BLACK
    g2D.drawLine(-(int)(0.2*osiWidth),0,
                  (int)(0.8*osiWidth),0);

    g2D.drawLine(0,-(int)(0.2*osiHeight),
                  0,(int)(0.8*osiHeight));

} //end setCoordinateFrame method
```

There is no intention to perform mathematical operations on the axes, so they are drawn independently of the classes and methods in the game-math library using the simplest method available for drawing a line.

The name of that simple method is **drawLine** , and it is a method of the standard Java **Graphics** class. The **translate** method is also a method of the **Graphics** class. Given that information, the code in [Listing 6](#) is straightforward and should not require further explanation.

Adding two vectors

Returning to the method named **drawOffScreen** , [Listing 7](#) begins by instantiating two objects of the **GM2D04.Vector** class.

Listing 7 . Adding two vectors.

```
GM2D04.Vector vecA = new GM2D04.Vector(  
                                new  
GM2D04.ColMatrix(50,100));  
GM2D04.Vector vecB = new GM2D04.Vector(  
                                new  
GM2D04.ColMatrix(75,25));  
  
GM2D04.Vector sumOf2 = vecA.add(vecB);
```

Then [Listing 7](#) calls the new **add** method (see [Listing 1](#)) on one of the vectors, passing the other vector as a parameter to the method. The **add** method returns a third vector that is the sum of the other two vectors. The new vector is referred to as **sumOf2** in [Listing 7](#).

Draw vecA in RED with its tail at the origin

Recall that a vector has only two properties: length and direction. It does not have a position property. Therefore, if you decide to draw a vector, you can draw it anywhere in space, and one position is equally as valid as any other position.

[Listing 8](#) sets the drawing color to RED and calls the **draw** method on **vecA** producing the red visual manifestation of the **Vector** object shown in [Figure 1](#).

(Note that there is also a magenta vector in [Figure 1](#), and it may be difficult to distinguish from the red vector, depending on the quality of the color on your monitor. The magenta vector is longer than the red vector.)

Listing 8 . Draw vecA in RED with its tail at the origin.

```
g2D.setColor(Color.RED);  
vecA.draw(g2D,new GM2D04.Point(  
new  
GM2D04.ColMatrix(0,0)));
```

Note that I elected to draw the vector with its tail at the origin, but that was an arbitrary decision that I will discuss in more detail later.

Draw vecB in GREEN head-to-tail with vecA

While it's legal to draw a vector anywhere in space, certain positions may have advantages over other positions in some cases. You will see what I mean shortly. In the meantime, [Listing 9](#) creates a visual manifestation of the **vecB** object with its tail at the head of the **vecA** object as shown by the green vector in [Figure 1](#).

Listing 9 . Draw vecB in GREEN head-to-tail with vecA.

```
g2D.setColor(Color.GREEN);  
vecB.draw(g2D,new GM2D04.Point(new  
GM2D04.ColMatrix(  
vecA.getData(0),vecA.getData(1))));
```

Draw sumOf2 in MAGENTA with its tail at the origin

[Listing 10](#) creates a visual manifestation of the **sumOf2** object with its tail at the origin as shown by the magenta vector in [Figure 1](#).

(In case you don't recognize the name of the color magenta, it looks similar to violet or purple. As mentioned earlier, it is somewhat longer than the red vector)

More correctly, [Listing 10](#) draws the **sumOf2** object with its tail coinciding with the tail of **vecA**. I elected to position the tails of the two vectors at the origin to keep the code a little simpler.

Listing 10 . Draw sumOf2 in MAGENTA with its tail at the origin.

```
g2D.setColor(Color.MAGENTA);
sumOf2.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
                0.0,0.0)));
```

The head-to-tail rule

We have just illustrated a very important rule that can make it much easier to visualize the results of vector addition. It is often called the *head-to-tail* rule.

If you add two or more vectors to produce a *sum* vector, and then draw the vectors that were included in the sum with the tail of one vector coinciding with the head of another vector, *(as illustrated by the red and green vectors in [Figure 1](#))*, the head of the last vector that you draw will be positioned at some particular point in space.

If you then draw the vector that represents the sum vector with its tail coinciding with the tail of the first vector that you drew, its head will coincide with the head of the last vector that you drew as shown by the magenta vector in [Figure 1](#).

The tail of the magenta vector coincides with the tail of the red vector, and the head of the magenta vector coincides with the head of the green vector.

(It doesn't matter whether or not the coinciding tails are drawn at the origin.)

Furthermore, this rule will hold regardless of the number of vectors included in the sum.

Extending the example to three vectors

[Listing 11](#) extends this example to three vectors.

Listing 11 . Extending the example to three vectors.

Listing 11 . Extending the example to three vectors.

```
//Now define another vector and add it to
vecA and
// vecB.
GM2D04.Vector vecC = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(30, -150));

//Draw vecD in BLUE with its tail
positioned at the
// sum of vecA and vecB
g2D.setColor(Color.BLUE);
vecC.draw(g2D, new GM2D04.Point(new
GM2D04.ColMatrix(
sumOf2.getData(0), sumOf2.getData(1))));

//Define a vector as the sum of vecA,
vecB, and vecC
GM2D04.Vector sumOf3 =
(vecA.add(vecB)).add(vecC);

//Draw sumOf3 in BLACK with its tail at
the origin.
g2D.setColor(Color.BLACK);
sumOf3.draw(g2D, new GM2D04.Point(
                                new
GM2D04.ColMatrix(0.0, 0.0)));

} //end drawOffScreen
```

You should understand this code

By now, you should have no difficulty understanding the code in [Listing 11](#) . The only tricky thing that I would call your attention to is the syntax of the code that adds the three vectors shown below for emphasis.

```
GM2D04.Vector sumOf3 = (vecA.add(vecB)).add(vecC);
```

You should make certain that you understand this syntax.

Note: No overloaded operators:

It is at times like this when I wish that Java supported overloaded operators in a manner similar to C++. Overloading the + operator would make the syntax much more intuitive than that shown by the code in [Listing 11](#) .

Back to the drawing

Now turn your attention back to the drawing in [Figure 1](#) . The red, green, and blue vectors are drawn in a head-to-tail manner as described earlier. The **sumOf3** (*black*) vector is drawn with its tail coinciding with the tail of the first (*red*) vector. Note that the head of the black vector coincides with the head of the last (*blue*) vector in the sum. Although not a definitive proof, this is at least a strong indication that the head-to-tail rule works for adding any number of vectors.

An overridden paint method

The **MyCanvas** class also includes an overridden **paint** method. However, the code in that method is very similar to code that I explained in the earlier module titled [GAME 2302-0110: Updating the Math Library for Graphics](#) . Therefore, I won't explain that code again in this module. You can view the overridden **paint** method in [Listing 20](#) .

That concludes the discussion of the program named **VectorAdd01** .

The program named `CoordinateFrame01`

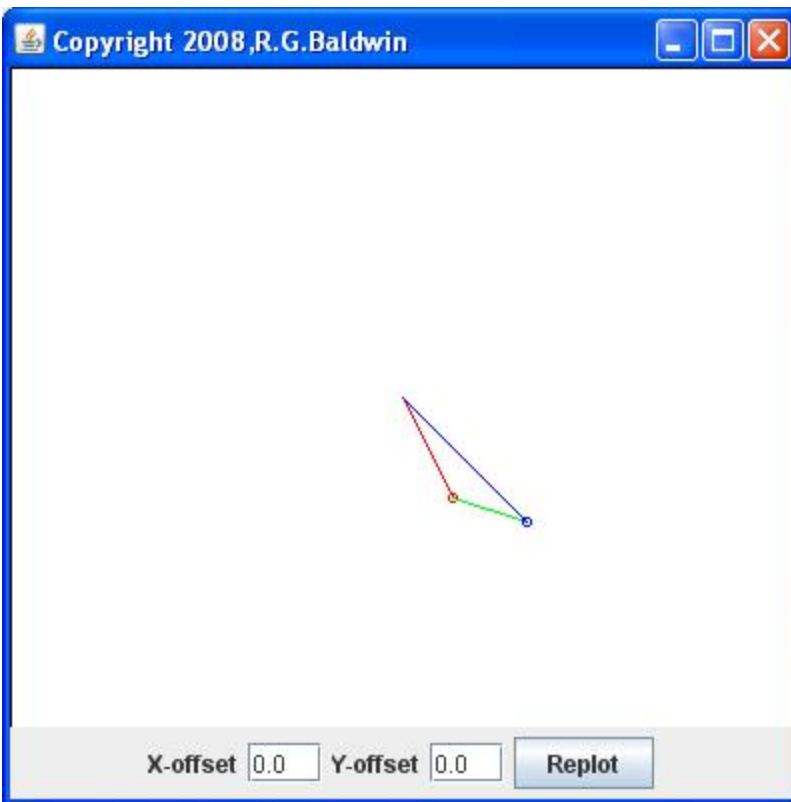
A complete listing of this program is provided in [Listing 21](#) near the end of the module. This program illustrates the relationship between the coordinate frame and a geometric object described in that coordinate frame.

A GUI allows the user to move the origin of the coordinate frame. Proper mathematical corrections are made such that a geometric object described in that coordinate frame maintains its position relative to world coordinates even when the coordinate frame is changed. This causes its position relative to the coordinate frame to change.

Screen output at startup

[Figure 2](#) shows the screen output of the program at startup.

Figure 2 Screen output from `CoordinateFrame01` at startup.



The axes are difficult to see

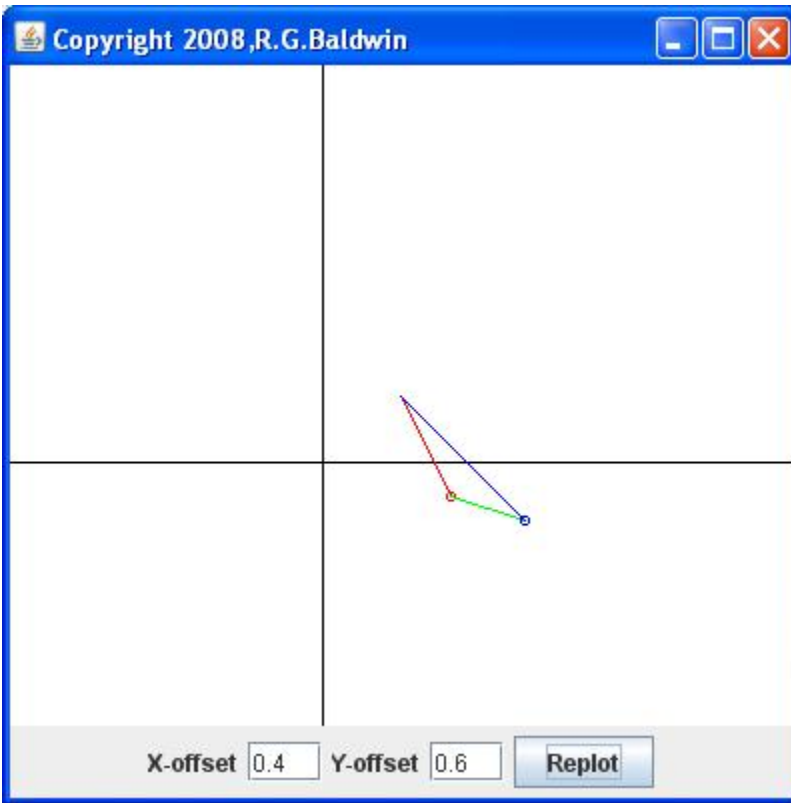
At this point, the origin is at the upper-left corner of the canvas. Although orthogonal axes are drawn intersecting at the origin, they are at the very edge of the canvas and are only barely visible. I will later refer to this coordinate frame with the origin in the upper-left corner as *world coordinates* .

A point has been defined (*but not drawn*) at a location specified by X equal to 196 and Y equal to 165. Even though the point is not drawn, a red vector has been drawn with its tail located at that point. A green vector has been drawn with its tail at the head of the red vector. A blue vector has been computed as the sum of the red and green vectors and it has been drawn with its tail at the tail of the red vector. As you learned in the previous program, this results in a closed polygon that is drawn at the point defined above.

Modify the coordinate frame

[Figure 3](#) shows the result of modifying the coordinate frame and then causing everything to be re-drawn. The coordinate frame is modified by entering values into the two user input fields and clicking the **Replot** button at the bottom of the GUI.

Figure 3 Screen output from CoordinateFrame01 after changes to the coordinate frame.



The new location of the origin

In [Figure 3](#), the origin of the coordinate frame has been moved to the right by an amount that is equal to 40-percent of the width of the off-screen image, and has been moved down by an amount that is 60-percent of the height of the off-screen image. Then a pair of orthogonal axes was drawn, intersecting at the new location of the origin.

A point is simply a location in space

Although a vector doesn't have a location property and therefore is immune to changes in the coordinate frame, a point does have a location property. In fact that is the only property that a point does have, because a point is simply a location in space. Furthermore, that location is always specified relative to some coordinate frame. In order to cause the point to remain in the same location relative to world coordinates, (*which was the objective here*), its values relative to the current coordinate frame must be modified each time the coordinate frame is modified.

After the values representing the point were modified appropriately, the three vectors were drawn in [Figure 3](#) with the tails of the red and blue vectors located at the point. This caused the geometric object described by the three vectors to remain in the same location relative to world coordinates. However, when the current coordinate frame no longer matched the world coordinate frame, the location of the geometric object changed relative to the current coordinate frame. The geometric object is closer to the origin of the current coordinate frame in [Figure 3](#) than was the case in [Figure 2](#).

More complicated code

The code in this program is somewhat more complicated than the code in the previous program, mainly due to the inclusion of an interactive GUI in the program. I have published many previous tutorials that explain how to write interactive programs in Java, and I won't repeat that explanation here. Instead, I will concentrate on the use of the game-math library in my explanation of this program. What this really means is that I am only going to explain the following three methods in this program:

- `actionPerformed`
- `setCoordinateFrame`
- `drawOffScreen`

You can view the remaining program code in [Listing 21](#) near the end of the module.

The `actionPerformed` method

This method must be defined in the class named **GUI** because the **GUI** class implements the **ActionListener** interface. Because an object of the **GUI** class is registered as a listener on the **Replot** button shown in [Figure 2](#), the **actionPerformed** method is called each time the user clicks the button.

Listing 12 . The actionPerformed method.

```
public void actionPerformed(ActionEvent e){
    //Reset the coordinate frame to world
coordinates by
    // reversing the most recent translation.
    setCoordinateFrame(g2D, -xOffset, -yOffset);

    //Compute new translation offsets based on
user input.
    xOffsetFactor =
Double.parseDouble(xOffsetField.getText());
    yOffsetFactor =
Double.parseDouble(yOffsetField.getText());

    xOffset = osiWidth*xOffsetFactor;
    yOffset = osiHeight*yOffsetFactor;

    //Draw a new off-screen image based on
user inputs
    // and copy it to the canvas. Note that
the
    // drawOffScreen method will call the
    // setCoordinateFrame method again with
the new
    // offset values for the origin of the
coordinate
    // frame.
    drawOffScreen(g2D);
    myCanvas.repaint();
} //end actionPerformed
```

What does the `actionPerformed` method do?

There is nothing complicated about the code in [Listing 12](#). This code performs the following actions as indicated by the embedded comments:

- Call the **`setCoordinateFrame`** method to reset the coordinate frame to world coordinates.
- Get user input values and use them to compute and save new offset values that will be used to define the origin of the new coordinate frame.
- Call the **`drawOffScreen`** method to erase the current image from the off-screen image and draw a new image on it using the new coordinate frame. Note that the **`drawOffScreen`** method will call the **`setCoordinateFrame`** method to establish the new coordinate frame before drawing the new image on the off-screen image.
- Call the **`repaint`** method to cause the off-screen image to be copied to the canvas and displayed on the computer screen.

The `setCoordinateFrame` method

This method, which is shown in [Listing 13](#), is used to set the coordinate frame of the off-screen image by setting the origin to the specified offset values relative to origin of the world coordinate frame.

Listing 13 . The `setCoordinateFrame` method.

Listing 13 . The setCoordinateFrame method.

```
private void setCoordinateFrame(  
                                Graphics2D g2D, double  
xOff, double yOff){  
  
    //Paint the background white  
    g2D.setColor(Color.WHITE);  
    g2D.fillRect(0,0,osiWidth,osiHeight);  
  
    //Translate the origin by the specified  
amount  
    g2D.translate((int)xOff,(int)yOff);  
  
    //Draw new X and Y-axes in BLACK  
    g2D.setColor(Color.BLACK);  
    g2D.drawLine(-(int)xOff,0,(int)(osiWidth-  
xOff),0);  
    g2D.drawLine(0,-(int)yOff,0,(int)  
((osiHeight-yOff)));  
  
} //end setCoordinateFrame method
```

What does the setCoordinateFrame method do?

As mentioned earlier, the origin of the world coordinate frame is the upper-left corner of the off-screen image. The **setCoordinateFrame** method assumes that the current coordinate frame coincides with the world coordinate frame when the method is called. This is because the method changes the coordinate frame relative to the current coordinate frame.

The method begins by painting the background white erasing anything already there. *(Note that this step erases the entire image only when the coordinate frame coincides with the world coordinate frame when the method is called.)*

Then the method establishes the new coordinate frame by translating the origin of the off-screen image using the X and Y offset values received as incoming parameters.

Finally, the method draws black orthogonal axes intersecting at the origin of the new coordinate frame on the off-screen image.

Beginning of the drawOffScreen method

The code for the method named **drawOffScreen** begins in [Listing 14](#). This method is called once in the constructor for the **GUI** class (see [Listing 21](#)), and again each time the user clicks the **Replot** button shown in [Figure 2](#).

The purpose of this method is to create some **Vector** objects and to cause visual manifestations of the **Vector** objects to be drawn onto an off-screen image.

Listing 14 . Beginning of the drawOffScreen method.

```
void drawOffScreen(Graphics2D g2D){  
    setCoordinateFrame(g2D,xOffset,yOffset);
```

The **drawOffScreen** method begins by calling the **setCoordinateFrame** method to establish a new coordinate frame using offset values entered by the user, *(or default offset values of 0.0 at startup)* .

Define a point to position the vectors

[Listing 15](#) instantiates a new **GM2D04.Point** object that will be used to locate the three vectors that form the closed polygon shown in [Figure 2](#).

The location of the polygon relative to the world coordinate frame will remain the same regardless of how the current coordinate frame is changed.

Listing 15 . Define a point to position the vectors.

```
double startPointX = (196 - xOffset);
double startPointY = (165 - yOffset);
GM2D04.Point startPoint = new
GM2D04.Point(
    new
    GM2D04.ColMatrix(startPointX, startPointY));
```

The code in [Listing 15](#) is probably the most important code in the entire program relative to the objective of the program. As I mentioned earlier, if you want the location of a point to remain the same relative to the world coordinate frame when you change the current coordinate frame, you must modify the values that represent that point whenever you cause the current coordinate frame to be different from the world coordinate frame. The code in [Listing 15](#) makes that modification.

Remaining code in the drawOffScreen method

The remaining code in the **drawOffScreen** method is shown in [Listing 16](#).

Listing 16 . Remaining code in the drawOffScreen method.

Listing 16 . Remaining code in the drawOffScreen method.

```
//Instantiate three Vector objects that
form a closed
// polygon when drawn head-to-tail.
double vecAx = 25;
double vecAy = 50;
GM2D04.Vector vecA = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(vecAx,vecAy));

double vecBx = 37;
double vecBy = -12;
GM2D04.Vector vecB = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(37,12));

//Define vecC as the sum of vecA and vecB.
It will be
// of the correct length and direction to
close the
// polygon when drawn such that its tail
coincides
// with the tail of vecA.
GM2D04.Vector vecC = vecA.add(vecB);

//Draw vecA in red with its tail at
startPoint.
g2D.setColor(Color.RED);
vecA.draw(g2D,startPoint);

//Compute the location of the head of vecA
relative to
// the current coordinate frame.
double headX =
                                vecA.getData(0) +
startPoint.getData(0);
```

Listing 16 . Remaining code in the drawOffScreen method.

```
double headY =
    vecA.getData(1) +
startPoint.getData(1);

//Draw vecB in GREEN with its tail at the
head of
// vecA.
g2D.setColor(Color.GREEN);
vecB.draw(g2D,new GM2D04.Point(
    new
GM2D04.ColMatrix(headX,headY)));

//Draw vecC in BLUE with its tail at
startPoint,
// coinciding with the tail of vecA. This
forms a
// closed polygon.
g2D.setColor(Color.BLUE);
vecC.draw(g2D,startPoint);

} //end drawOffScreen
```

With one exception, there is nothing in [Listing 16](#) that I haven't already explained in earlier programs in this or previous modules. Therefore, I won't repeat those explanations here.

The one exception

Recall from [Figure 2](#) that we need to draw the green vector with its tail located at the head of the red vector. In order to do that, we must be able to determine the location of the head of the red vector relative to the current coordinate frame.

The **getData** method of the **Vector** class knows nothing about position. That method simply returns the X and Y coordinate values that describe the length of the vector relative to its tail.

*(For the special case where the tail is located at the origin, the **getData** method returns the X and Y coordinates of the head of the vector.)*

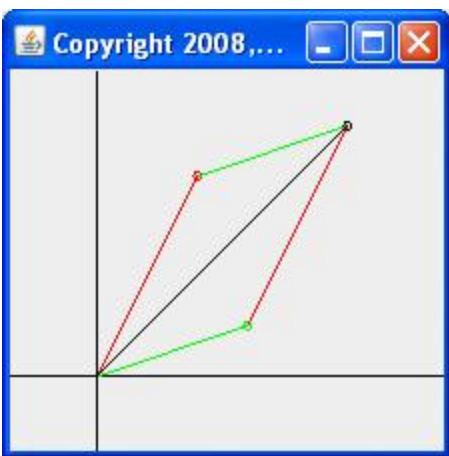
However, because the tail of the red vector in this case is not necessarily located at the origin, we must calculate the position of the head of the red vector taking the position of its tail (**startPoint**) into account. The code in [Listing 16](#) does just that.

That concludes the discussion of the program named **CoordinateFrame01** .

The program named VectorAdd02

Hopefully, you have been studying the Kjell tutorial as instructed in the section titled [Homework assignment](#) . This program illustrates the addition of two vectors using two different approaches that result in the parallelogram described by Kjell. That parallelogram is shown in the screen output in [Figure 4](#) .

Figure 4 Screen output from the program named VectorAdd02.



The method named **drawOffScreen**

A complete listing of this program is provided in [Listing 22](#) near the end of the module. Most of the new material is contained in the method named **drawOffScreen** . Therefore, I will limit my explanation to the method named **drawOffScreen** , which begins in [Listing 17](#) .

Listing 17 . Beginning of the method named drawOffScreen of the program named VectorAdd02.

```
void drawOffScreen(Graphics2D g2D){

    //Translate the origin to a position near
the bottom-
    // left corner of the off-screen image and
draw a pair
    // of orthogonal axes that intersect at
the origin.
    setCoordinateFrame(g2D);

    //Define two vectors.
    GM2D04.Vector vecA = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(50, -100));
    GM2D04.Vector vecB = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(75, -25));

    //Draw vecA in RED with its tail at the
origin
    g2D.setColor(Color.RED);
    vecA.draw(g2D, new GM2D04.Point(
                                new
```

Listing 17 . Beginning of the method named drawOffScreen of the program named VectorAdd02.

```
GM2D04.ColMatrix(0,0)));

    //Draw vecB in GREEN with its tail at the
    head of vecA
    g2D.setColor(Color.GREEN);
    vecB.draw(g2D,new GM2D04.Point(new
    GM2D04.ColMatrix(

vecA.getData(0),vecA.getData(1))));

    //Define a third vector as the sum of the
    first
    // two vectors defined above by adding
    vecB to vecA.
    GM2D04.Vector sumA = vecA.add(vecB);

    //Draw sumA in BLACK with its tail at the
    origin.
    // The head will coincide with the head of
    the
    // green vecB.
    g2D.setColor(Color.BLACK);
    sumA.draw(g2D,new GM2D04.Point(
                                new
    GM2D04.ColMatrix(0.0,0.0)));
```

There is nothing new in [Listing 17](#). The code in [Listing 17](#) produces the black vector in [Figure 4](#) plus the red and green vectors that appear above the black vector.

Do the same operations in a different order

[Listing 18](#) begins by drawing the red and green vectors again. However, this time the green vector is drawn with its tail at the origin, and the red vector is drawn with its tail at the head of the green vector as shown by the green and red vectors below the black vector in [Figure 4](#). This is perfectly legal because a vector has no location property. We can draw a vector anywhere we please provided we draw it with the correct length and direction.

Listing 18 . Do the same operations in a different order.

```
//Draw vecB in GREEN with its tail at the
origin.
g2D.setColor(Color.GREEN);
vecB.draw(g2D,new GM2D04.Point(
                                new
GM2D04.ColMatrix(0,0)));

//Draw vecA in RED with its tail at the
head of vecB.
g2D.setColor(Color.RED);
vecA.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
vecB.getData(0),vecB.getData(1))));

//Define a fourth vector as the sum of the
first
// two vectors defined above by adding
vecA to vecB.
GM2D04.Vector sumB = vecB.add(vecA);

//Draw sumB in BLACK with its tail at the
```

Listing 18 . Do the same operations in a different order.

```
origin.  
    // The head will coincide with the head of  
the  
    // red vecA, and the visual manifestation  
of sumB will  
    // overlay the visual manifestation of  
sumA  
    g2D.setColor(Color.BLACK);  
    sumB.draw(g2D,new GM2D04.Point(  
                                                new  
GM2D04.ColMatrix(0.0,0.0)));  
  
    }//end drawOffScreen
```

Then [Listing 18](#) creates another vector by adding the red and green vectors and refers to the new vector as **sumB** . In the case of Listing 17, **vecB** is added to **vecA** to produce **sumA** . In Listing 18, **vecA** is added to **vecB** to produce **sumB** . In other words, the two addition operations are performed in reverse order in the two listings. As I have mentioned before, the order in which you add vectors doesn't matter. The result will be the same no matter the order in which you add them.

This is demonstrated in [Listing 18](#) by drawing the vector referred to by **sumB** in black. As you can see in [Figure 4](#), the drawing of **sumB** overlays the earlier drawing of **sumA** and you can't tell one from the other.

*(You know that the length of the two vectors is the same because the little circles at the heads of the two vectors overlay one another. You know that the directions are the same because **sumB** completely covers **sumA** .)*

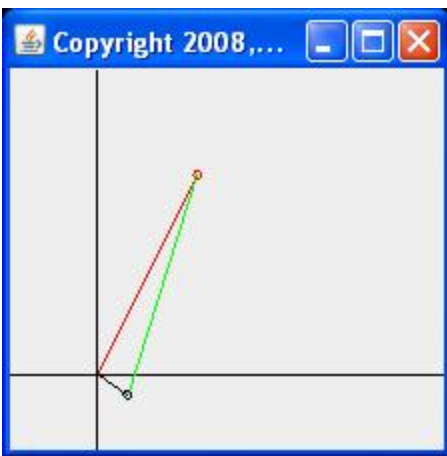
That concludes the discussion of the program named **VectorAdd02** .

The program named VectorAdd03

This program illustrates the fact that the length of the sum of two vectors is not necessarily equal to the sum of the lengths of the two vectors.

Two vectors are added by the program. The length of the sum is much smaller than the length of either of the original vectors. This is shown in [Figure 5](#) where the red vector is added to the green vector producing the shorter black vector.

Figure 5 Graphic screen output from the program named VectorAdd03.



Command-line output

In addition to the graphic output shown in [Figure 5](#), this program also produces three lines of text on the command-line screen as shown in [Figure 6](#).

Figure 6 . Command-line output from the program named VectorAdd03.

Figure 6 . Command-line output from the program named VectorAdd03.

```
Red length = 111.80339887498948  
Green length = 115.43396380615195  
Black length = 18.027756377319946
```

The text output on the command-line screen shows the length of each vector. As you can see in [Figure 6](#), the length of the black vector is much less than the length of either the red vector or the blue vector.

Not much that is new

The only thing that is new in this program is the call to the new **getLength** method of the **GM2D04.Vector** class to compute and display the length of each vector in [Figure 6](#).

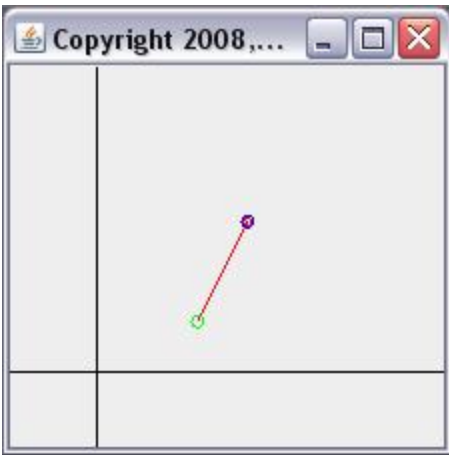
The source code for the **getLength** method is shown in [Listing 2](#). This code computes the length of the vector as the square root of the sum of the squares of the X and Y components of the vector. Other than that code, there is no code in this program that warrants further explanation.

You can view a complete listing of the program in [Listing 23](#) near the end of the module.

The program named VectorAdd04

This program illustrates the addition of a **Vector** object to a **Point** object producing a new **Point** object. Both points and the vector are drawn on the screen as shown in [Figure 7](#).

Figure 7 Screen output from the program named VectorAdd04.



The only thing that is new in this program is the call to the new **addVectorToPoint** method of the **GM2D04.Point** class to add the vector to the point producing a new point.

The source code for the **addVectorToPoint** method is shown in [Listing 3](#). You can view a complete listing of the program in [Listing 24](#) near the end of the module.

A very powerful capability

Don't be fooled by the apparent simplicity of the **addVectorToPoint** method. The ability to add a vector to a point provides a powerful new capability to the game-math library. As you will see in the next module, this capability makes it possible not only to translate geometrical objects from one location in space to another, but also makes it possible to animate geometrical objects.

Documentation for the GM2D04 library

Click [here](#) to download a zip file containing standard javadoc documentation for the library named **GM2D04**. Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text (see [Listing 19](#) and the explanations given above), the

documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter 3 - Vector Addition*.

The homework assignment for the next module is to study the Kjell tutorial through *Chapter 5 - Vector Direction*.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 19](#) through [Listing 24](#). Compile the code and execute it in conjunction with the game-math library provided in [Listing 19](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module you learned

- How to add two or more vectors
- About the head-to-tail rule in vector addition
- About the vector addition parallelogram
- About the relationship between the length of the sum of vectors and the lengths of the individual vectors in the sum

- How to add a vector to a point
- How to get the length of a vector
- How to represent an object in different coordinate frames

What's next?

In the next module you will learn how to use the game-math library for translation and animation in two dimensions.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0125: Vector Addition
- File: Game0125.htm
- Published: 10/15/12
- Revised: 02/02/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a

book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 19](#) through [Listing 24](#) below.

Listing 19 . Source code for the game-math library named GM2D04.

```
/*GM2D04.java  
Copyright 2008, R.G.Baldwin  
Revised 02/08/08
```

The name GM2Dnn is an abbreviation for GameMath2Dnn.

See the file named GM2D01.java for a general description of this game-math library file. This file is an update of GM2D03.

This update added the following new capabilities:

Vector addition - Adds this Vector object to a Vector object received as an incoming parameter and returns the sum as a resultant Vector object.

Added a method named getLength that returns the length

of a vector as type double.

Added a method named addVectorToPoint to add a Vector to a Point producing a new Point.

Tested using JDK 1.6 under WinXP.

```
*****
*****/
import java.awt.geom.*;
import java.awt.*;

public class GM2D04{

    //An object of this class represents a 2D column
    matrix.
    // An object of this class is the fundamental
    building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix{
        double[] data = new double[2];

        public ColMatrix(double data0,double data1){
            data[0] = data0;
            data[1] = data1;
        }//end constructor
        //-----
    }

    -----//

    public String toString(){
        return data[0] + "," + data[1];
    }//end overridden toString method
    //-----
    -----//

    public double getData(int index){
```

```

        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return data[index];
        } //end else
    } //end getData method
//-----
-----//

```

```
public void setData(int index, double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        this.data[index] = data;
    } //end else
} //end setData method
//-----
-----//
```

```
//This method overrides the equals method
inherited
// from the class named Object. It compares
the values
// stored in two matrices and returns true if
the
// values are equal or almost equal and
returns false
// otherwise.
public boolean equals(Object obj){
    if(obj instanceof GM2D04.ColMatrix &&
Math.abs(((GM2D04.ColMatrix)obj).getData(0) -
        getData(0)) <=
0.000001 &&
Math.abs(((GM2D04.ColMatrix)obj).getData(1) -
        getData(1)) <=
```



```

0.00001){
    return true;
}else{
    return false;
} //end else

} //end overridden equals method
//-----
-----//

    //Adds one ColMatrix object to another
ColMatrix
    // object, returning a ColMatrix object.
    public GM2D04.ColMatrix add(GM2D04.ColMatrix
matrix){
    return new GM2D04.ColMatrix(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1));
    } //end subtract
    //-----
    -----//

    //Subtracts one ColMatrix object from another
    // ColMatrix object, returning a ColMatrix
object. The
    // object that is received as an incoming
parameter
    // is subtracted from the object on which the
method
    // is called.
    public GM2D04.ColMatrix subtract(
                                GM2D04.ColMatrix
matrix){
    return new GM2D04.ColMatrix(
                                getData(0)-

```

```

matrix.getData(0),
                                getData(1)-
matrix.getData(1));
    }//end subtract
    //-----
-----//
    }//end class ColMatrix

//=====
====//

    public static class Point{
        GM2D04.ColMatrix point;

        public Point(GM2D04.ColMatrix point)
    {//constructor
        //Create and save a clone of the ColMatrix
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix
object
        // through use of its set method.
        this.point =
            new
ColMatrix(point.getData(0),point.getData(1));
    }//end constructor
    //-----
-----//

    public String toString(){
        return point.getData(0) + "," +
point.getData(1);
    }//end toString
    //-----

```

-----//

```
public double getData(int index){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        return point.getData(index);
    }//end else
}//end getData
```

//-----

-----//

```
public void setData(int index,double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        point.setData(index,data);
    }//end else
}//end setData
```

//-----

-----//

//This method draws a small circle around the
location
// of the point on the specified graphics
context.

```
public void draw(Graphics2D g2D){
    Ellipse2D.Double circle =
        new
    Ellipse2D.Double(getData(0)-3,
    getData(1)-3,
```

6,
6);

```
    g2D.draw(circle);
}//end draw
```

//-----

```

-----//

    //Returns a reference to the ColMatrix object
that
    // defines this Point object.
    public GM2D04.ColMatrix getColMatrix(){
        return point;
    }//end getColMatrix
    //-----
-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix objects that define
two
    // Point objects and returns true if they are
equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(point.equals(((GM2D04.Point)obj).
getColMatrix())){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
-----//

    //Gets a displacement vector from one Point
object to
    // a second Point object. The vector points
from the

```

```

        // object on which the method is called to the
object
        // passed as a parameter to the method. Kjell
        // describes this as the distance you would
have to
        // walk along the x and then the y axes to get
from
        // the first point to the second point.
        public GM2D04.Vector getDisplacementVector(
                                                GM2D04.Point
point){
            return new GM2D04.Vector(new
GM2D04.ColMatrix(
                                                point.getData(0)-
getData(0),
                                                point.getData(1)-
getData(1)));
        }//end getDisplacementVector
        //-----
        -----//

        //Adds a Vector to a Point producing a new
Point.
        public GM2D04.Point addVectorToPoint(
GM2D04.Vector vec){
            return new GM2D04.Point(new
GM2D04.ColMatrix(
                                                getData(0) +
vec.getData(0),
                                                getData(1) +
vec.getData(1)));
        }//end addVectorToPoint
        //-----
        -----//
    }//end class Point

```

```

//=====
====//

    public static class Vector{
        GM2D04.ColMatrix vector;

        public Vector(GM2D04.ColMatrix vector)
{//constructor
    //Create and save a clone of the ColMatrix
object
    // used to define the vector to prevent the
vector
    // from being corrupted by a later change in
the
    // values stored in the original ColVector
object.
    this.vector = new ColMatrix(
vector.getData(0),vector.getData(1));
    }//end constructor
    //-----
    -----//

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1);
    }//end toString
    //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData

```

```

//-----
-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData
//-----
-----//

```

```

//This method draws a vector on the specified
graphics
// context, with the tail of the vector
located at a
// specified point, and with a small circle at
the
// head.

```

```

    public void draw(Graphics2D g2D,GM2D04.Point
tail){
        Line2D.Double line = new Line2D.Double(
                                tail.getData(0),
                                tail.getData(1),

```

```

tail.getData(0)+vector.getData(0),

```

```

tail.getData(1)+vector.getData(1));

```

```

        //Draw a small circle to identify the head.
        Ellipse2D.Double circle = new
Ellipse2D.Double(

```

```

tail.getData(0)+vector.getData(0)-2,

```

```

tail.getData(1)+vector.getData(1)-2,

```

```

                                4,
                                4);
        g2D.draw(circle);
        g2D.draw(line);
    }//end draw
    //-----
-----//

    //Returns a reference to the ColMatrix object
that
    // defines this Vector object.
    public GM2D04.ColMatrix getColMatrix(){
        return vector;
    }//end getColMatrix
    //-----
-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix objects that define
two
    // Vector objects and returns true if they are
equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(vector.equals((
(GM2D04.Vector)obj).getColMatrix())){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----

```



```

-----//

    //Adds this vector to a vector received as an
incoming
    // parameter and returns the sum as a vector.
    public GM2D04.Vector add(GM2D04.Vector vec){
        return new GM2D04.Vector(new ColMatrix(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1)));
    }//end add
    //-----
-----//

    //Returns the length of a Vector object.
    public double getLength(){
        return Math.sqrt(
            getData(0)*getData(0) +
            getData(1)*getData(1));
    }//end getLength
    //-----
-----//
    }//end class Vector

//=====
=====//

```

```

    //A line is defined by two points. One is called
the
    // tail and the other is called the head.
    public static class Line{
        GM2D04.Point[] line = new GM2D04.Point[2];

        public Line(GM2D04.Point tail,GM2D04.Point
head){

```

```

        //Create and save clones of the points used
to
        // define the line to prevent the line from
being
        // corrupted by a later change in the
coordinate
        // values of the points.
        this.line[0] = new Point(new
GM2D04.ColMatrix(

tail.getData(0),tail.getData(1)));
        this.line[1] = new Point(new
GM2D04.ColMatrix(

head.getData(0),head.getData(1)));
    }//end constructor
    //-----
    -----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + ","
            + line[1].getData(1);
    }//end toString
    //-----
    -----//

    public GM2D04.Point getTail(){
        return line[0];
    }//end getTail
    //-----
    -----//

    public GM2D04.Point getHead(){
        return line[1];
    }//end getHead

```

```

//-----
-----//

    public void setTail(GM2D04.Point newPoint){
        //Create and save a clone of the new point
to        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the      // point.
        this.line[0] = new Point(new
GM2D04.ColMatrix(
newPoint.getData(0),newPoint.getData(1)));
        }//end setTail
        //-----
-----//

```

```

    public void setHead(GM2D04.Point newPoint){
        //Create and save a clone of the new point
to        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the      // point.
        this.line[1] = new Point(new
GM2D04.ColMatrix(
newPoint.getData(0),newPoint.getData(1)));
        }//end setHead
        //-----
-----//

```

```

    public void draw(Graphics2D g2D){
        Line2D.Double line = new Line2D.Double(

```

```

getTail().getData(0),
getTail().getData(1),
getHead().getData(0),
getHead().getData(1));
    g2D.draw(line);
} //end draw
//-----
-----//
} //end class Line

//=====
====//

} //end class GM2D04
//=====
=====//

```

Listing 20 . Source code for the program named VectorAdd01.

```

/*VectorAdd01.java
Copyright 2008, R.G.Baldwin
Revised 02/10/08

```

This program illustrates the addition of two or more vectors. It also illustrates the Head-to-Tail rule described by Kjell.

Tested using JDK 1.6 under WinXP.

```

*****
*****/
import java.awt.*;
import javax.swing.*;

```

```

import java.awt.geom.*;

class VectorAdd01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class VectorAdd01
//=====
=====//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 275;
    int vSize = 200;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to
        the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        //This object must be visible before you can
        get an
        // off-screen image. It must also be visible

```

before

```
// you can compute the size of the canvas.
setVisible(true);
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

//Create an off-screen image and get a
graphics
// context on it.
osi = createImage(osiWidth,osiHeight);
Graphics2D g2D = (Graphics2D)
(osi.getGraphics());

//Draw some graphical objects on the off-
screen
// image that represent underlying data
objects in
// 2D space.
drawOffScreen(g2D);

//Cause the overridden paint method belonging
to
// myCanvas to be executed.
myCanvas.repaint();

} //end constructor
//-----
-----//

//The purpose of this method is to add some
Vector
// objects and to cause visual manifestations of
the raw
// Vector objects and the resultant Vector
objects to be
// drawn onto an off-screen image.
void drawOffScreen(Graphics2D g2D){
```

```

    //Translate the origin on the off-screen
    // image and draw a pair of orthogonal axes on
it.
    setCoordinateFrame(g2D);

    //Define two vectors.
    GM2D04.Vector vecA = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(50,100));
    GM2D04.Vector vecB = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(75,25));

    //Define a third vector as the sum of the
first
    // two vectors defined above.
    GM2D04.Vector sumOf2 = vecA.add(vecB);

    //Draw vecA in RED with its tail at the origin
    g2D.setColor(Color.RED);
    vecA.draw(g2D,new GM2D04.Point(
                                new
GM2D04.ColMatrix(0,0)));

    //Draw vecB in GREEN with its tail at the head
of vecA
    g2D.setColor(Color.GREEN);
    vecB.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
vecA.getData(0),vecA.getData(1))));

    //Draw sumOf2 in MAGENTA with its tail at the
origin.
    // The head will coincide with the head of
vecB.

```

```

        g2D.setColor(Color.MAGENTA);
        sumOf2.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
                                0.0,0.0)));

        //Now define another vector and add it to vecA
and
        // vecB.
        GM2D04.Vector vecC = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(30,-150));

        //Draw vecD in BLUE with its tail positioned
at the
        // sum of vecA and vecB
        g2D.setColor(Color.BLUE);
        vecC.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
sumOf2.getData(0),sumOf2.getData(1))));

        //Define a vector as the sum of vecA, vecB,
and vecC
        GM2D04.Vector sumOf3 =
        (vecA.add(vecB)).add(vecC);

        //Draw sumOf3 in BLACK with its tail at the
origin.
        g2D.setColor(Color.BLACK);
        sumOf3.draw(g2D,new GM2D04.Point(
                                new
GM2D04.ColMatrix(0.0,0.0)));

        }//end drawOffScreen
        //-----
        -----//

```



```

    //This method is used to set the origin to a
point near
    // the upper-left corner of the off-screen image
and
    // draw orthogonal axes on the off-screen image.
There
    // is no intention to perform mathematical
operations on
    // the axes, so they are drawn independently of
the
    // classes and methods in the game-math library
using
    // the simplest method available for drawing a
line.
    private void setCoordinateFrame(Graphics2D g2D){
        //Translate the origin.
        g2D.translate(0.2*osiWidth,0.2*osiHeight);

        //Draw new X and Y-axes in default BLACK
        g2D.drawLine(-(int)(0.2*osiWidth),0,
                    (int)(0.8*osiWidth),0);

        g2D.drawLine(0,-(int)(0.2*osiHeight),
                    0,(int)(0.8*osiHeight));

    }//end setCoordinateFrame method

//=====
====//

```

```

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the

```

```

        // screen or when the repaint method is called
on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
//=====
=====//

```

Listing 21 . Source code for the program named CoordinateFrame01.

```

/*CoordinateFrame01.java
Copyright 2008, R.G.Baldwin
Revised 02/14/08

```

This program illustrates the relationship between the coordinate frame and a geometric object described in that coordinate frame.

A GUI allows the user to move the origin of the coordinate frame. Proper mathematical corrections are made such that a geometric object described in that coordinate frame maintains its position relative to world coordinates even when the coordinate frame is changed. This causes its

position relative to the coordinate frame to change.

Tested using JDK 1.6 under WinXP.

*****/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;
```

```
class CoordinateFrame01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class CoordinateFrame01
//=====
=====//
```

```
class GUI extends JFrame implements
ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    //The following offset values are applied to the
    width
    // and the height of the off-screen image to
    modify the
    // coordinate frame. They are used to store user
    input
```

```

// values.
double xOffsetFactor = 0.0;
double yOffsetFactor = 0.0;

//The following offset values are computed using
the
// user-specified offset factor values.
double xOffset;
double yOffset;

Graphics2D g2D;//Off-screen graphics context.

//User input fields.
JTextField xOffsetField;
JTextField yOffsetField;
//-----
-----//

GUI(){//constructor
    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create a new drawing canvas and add it to
the
    // center of the content pane.
    myCanvas = new MyCanvas();
    this.getContentPane().add(

BorderLayout.CENTER,myCanvas);

    //Create and populate a JPanel to be used as a
user-
    // input panel and add it to the SOUTH
position on

```

```

// the content pane.
JPanel controlPanel = new JPanel();
controlPanel.add(new JLabel("X-offset"));
xOffsetField = new JTextField("0.0",3);
controlPanel.add(xOffsetField);

controlPanel.add(new JLabel("Y-offset"));
yOffsetField = new JTextField("0.0",3);
controlPanel.add(yOffsetField);

JButton button = new JButton("Replot");
controlPanel.add(button);
this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

//Register this object as an action listener
on the
// button.
button.addActionListener(this);

//This object must be visible before you can
get an
// off-screen image. It must also be visible
before
// you can compute the size of the canvas.
setVisible(true);

//Make the size of the off-screen image match
the
// size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

//Create an off-screen image and get a
graphics
// context on it.

```

```

    osi = createImage(osiWidth,osiHeight);
    g2D = (Graphics2D)(osi.getGraphics());

    //Create some underlying data objects in
    // 2D space and draw visual manifestations of
them.
    drawOffScreen(g2D);

    //Cause the overridden paint method belonging
to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
//-----
-----//

    //The purpose of this method is to create some
Vector
    // objects and to cause visual manifestations of
them to
    // be drawn onto an off-screen image.
    void drawOffScreen(Graphics2D g2D){

        //Establish the current coordinate frame and
prepare
        // the off-screen image with axes, etc.
        setCoordinateFrame(g2D,xOffset,yOffset);

        //Define a Point that will be used to locate
three
        // vectors that form a closed polygon. The
physical
        // location of the polygon on the canvas
(world)
        // remains the same regardless of how the
coordinate

```

```

// frame is changed.
double startPointX = (196 - xOffset);
double startPointY = (165 - yOffset);
GM2D04.Point startPoint = new GM2D04.Point(
    new
GM2D04.ColMatrix(startPointX, startPointY));

//Instantiate three Vector objects that form a
closed
// polygon when drawn head-to-tail.
double vecAx = 25;
double vecAy = 50;
GM2D04.Vector vecA = new GM2D04.Vector(
    new
GM2D04.ColMatrix(vecAx, vecAy));

double vecBx = 37;
double vecBy = -12;
GM2D04.Vector vecB = new GM2D04.Vector(
    new
GM2D04.ColMatrix(37, 12));

//Define vecC as the sum of vecA and vecB. It
will be
// of the correct length and direction to
close the
// polygon when drawn such that its tail
coincides
// with the tail of vecA.
GM2D04.Vector vecC = vecA.add(vecB);

//Draw vecA in red with its tail at
startPoint.
g2D.setColor(Color.RED);
vecA.draw(g2D, startPoint);

//Compute the location of the head of vecA

```

```

relative to
    // the current coordinate frame.
    double headX =
        vecA.getData(0) +
startPoint.getData(0);
    double headY =
        vecA.getData(1) +
startPoint.getData(1);

    //Draw vecB in GREEN with its tail at the head
of
    // vecA.
    g2D.setColor(Color.GREEN);
    vecB.draw(g2D,new GM2D04.Point(
        new
GM2D04.ColMatrix(headX,headY)));

    //Draw vecC in BLUE with its tail at
startPoint,
    // coinciding with the tail of vecA. This
forms a
    // closed polygon.
    g2D.setColor(Color.BLUE);
    vecC.draw(g2D,startPoint);

} //end drawOffScreen
//-----
-----//

    //This method is used to set the coordinate
frame of
    // the off-screen image by setting the origin to
the
    // specified offset values relative to origin of
the
    // world. The origin of the world is the upper-
left

```



```

// corner of the off-screen image.
//The method draws black orthogonal axes on the
// off-screen image.
//There is no intention to perform mathematical
// operations on the axes, so they are drawn
// independently of the classes and methods in
the
// game-math library.
//The method paints the background white erasing
// anything already there.
private void setCoordinateFrame(
                                Graphics2D g2D,double
xOff,double yOff){

    //Paint the background white
    g2D.setColor(Color.WHITE);
    g2D.fillRect(0,0,osiWidth,osiHeight);

    //Translate the origin by the specified amount
    g2D.translate((int)xOff,(int)yOff);

    //Draw new X and Y-axes in BLACK
    g2D.setColor(Color.BLACK);
    g2D.drawLine(-(int)xOff,0,(int)(osiWidth-
xOff),0);
    g2D.drawLine(0,-(int)yOff,0,(int)((osiHeight-
yOff)));

    }//end setCoordinateFrame method
    //-----
    -----//

    //This method must be defined because the class
    // implements the ActionListener interface.
Because an
    // object of this class is registered as a
listener on

```

```

    // the button, this method is called each time
the user
    // presses the button.
    public void actionPerformed(ActionEvent e){
        //Reset the coordinate frame to world
coordinates by
        // reversing the most recent translation.
        setCoordinateFrame(g2D, -xOffset, -yOffset);

        //Compute new translation offsets based on
user input.
        xOffsetFactor =
Double.parseDouble(xOffsetField.getText());
        yOffsetFactor =
Double.parseDouble(yOffsetField.getText());

        xOffset = osiWidth*xOffsetFactor;
        yOffset = osiHeight*yOffsetFactor;

        //Draw a new off-screen image based on user
inputs
        // and copy it to the canvas. Note that the
        // drawOffScreen method will call the
        // setCoordinateFrame method again with the
new
        // offset values for the origin of the
coordinate
        // frame.
        drawOffScreen(g2D);
        myCanvas.repaint();
    }//end actionPerformed

//=====
====//

```

```

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the
        // screen or when the repaint method is called
on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
//=====
=====//

```

Listing 22 . Source code for the program named VectorAdd02.

```

/*VectorAdd02.java
Copyright 2008, R.G.Baldwin
Revised 02/14/08

```

This program illustrates the addition of two vectors using two different approaches that result in the parallelogram described by Kjell. It also illustrates the Head-to-Tail rule described by Kjell.

Tested using JDK 1.6 under WinXP.

```

*****
*****/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

class VectorAdd02{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class VectorAdd02
//=====
=====//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 225;
    int vSize = 225;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to
        the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);
    }
}

```

```

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth,osiHeight);
Graphics2D g2D = (Graphics2D)
(osi.getGraphics());

        //Draw some graphical objects on the off-
screen
        // image that represent underlying data
objects in
        // 2D space.
drawOffScreen(g2D);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.
myCanvas.repaint();

    }//end constructor
    //-----
    -----//

        //The purpose of this method is to add some
Vector
        // objects and to cause visual manifestations of
the raw

```

```

    // Vector objects and the resultant Vector
objects to be
    // drawn onto an off-screen image.
    void drawOffScreen(Graphics2D g2D){

        //Translate the origin to a position near the
bottom-
        // left corner of the off-screen image and
draw a pair
        // of orthogonal axes that intersect at the
origin.
        setCoordinateFrame(g2D);

        //Define two vectors.
        GM2D04.Vector vecA = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(50, -100));
        GM2D04.Vector vecB = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(75, -25));

        //Draw vecA in RED with its tail at the origin
        g2D.setColor(Color.RED);
        vecA.draw(g2D, new GM2D04.Point(
                                new
GM2D04.ColMatrix(0, 0)));

        //Draw vecB in GREEN with its tail at the head
of vecA
        g2D.setColor(Color.GREEN);
        vecB.draw(g2D, new GM2D04.Point(new
GM2D04.ColMatrix(
vecA.getData(0), vecA.getData(1))));

        //Define a third vector as the sum of the
first

```

```
// two vectors defined above by adding vecB to  
vecA.
```

```
GM2D04.Vector sumA = vecA.add(vecB);
```

```
//Draw sumA in BLACK with its tail at the  
origin.
```

```
// The head will coincide with the head of the  
// green vecB.
```

```
g2D.setColor(Color.BLACK);
```

```
sumA.draw(g2D,new GM2D04.Point(  
                                new  
GM2D04.ColMatrix(0.0,0.0)));
```

```
//Do the same operations but in a different  
order.
```

```
//Draw vecB in GREEN with its tail at the  
origin.
```

```
g2D.setColor(Color.GREEN);
```

```
vecB.draw(g2D,new GM2D04.Point(  
                                new  
GM2D04.ColMatrix(0,0)));
```

```
//Draw vecA in RED with its tail at the head  
of vecB.
```

```
g2D.setColor(Color.RED);
```

```
vecA.draw(g2D,new GM2D04.Point(new  
GM2D04.ColMatrix(  
vecB.getData(0),vecB.getData(1))));
```

```
//Define a fourth vector as the sum of the  
first
```

```
// two vectors defined above by adding vecA to  
vecB.
```

```
GM2D04.Vector sumB = vecB.add(vecA);
```

```

        //Draw sumB in BLACK with its tail at the
origin.
        // The head will coincide with the head of the
        // red vecA, and the visual manifestation of
sumB will
        // overlay the visual manifestation of sumA
        g2D.setColor(Color.BLACK);
        sumB.draw(g2D,new GM2D04.Point(
                                new
GM2D04.ColMatrix(0.0,0.0)));

    }//end drawOffScreen
    //-----
    -----//

    //This method is used to set the origin of the
    // off-screen image to a location near the
lower-left
    // corner and to draw orthogonal axes that
intersect
    // at the origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Translate the origin to a point near the
lower-left
        // corner of the off-screen image.
        g2D.translate(0.2*osiWidth,0.8*osiHeight);

        //Draw new X and Y-axes in default BLACK
        g2D.drawLine(-(int)(0.2*osiWidth),0,
                    (int)(0.8*osiWidth),0);

        g2D.drawLine(0,-(int)(0.8*osiHeight),
                    0,(int)(0.2*osiHeight));

    }//end setCoordinateFrame method

```



```
//=====
====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the
        // screen or when the repaint method is called
on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI
//=====
====//
```

Listing 23 . Source code for the program named VectorAdd03.

```
/*VectorAdd03.java
Copyright 2008, R.G.Baldwin
Revised 02/14/08
```

This program illustrates the fact that the length of the sum of two vectors is not necessarily equal to the sum of the lengths of the two vectors. Two vectors are added such and the length of the sum is much smaller than the

length
of either of the original vectors. The red and
green
vectors shown in the screen output are added
producing the
black vector. The lengths of all three vectors are
displayed on the command-line screen,
demonstrating that
the length of the black vector is much less than
the
length of either the red vector or the blue
vector.

The only thing that is new in this program is the
getLength method of the GM2D04.Vector class.

Tested using JDK 1.6 under WinXP.

*****/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
```

```
class VectorAdd03{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class VectorAdd03
//=====
=====//
```

```
class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 225;
    int vSize = 225;
```

```

Image osi;//an off-screen image
int osiWidth;//off-screen image width
int osiHeight;//off-screen image height
MyCanvas myCanvas;//a subclass of Canvas

GUI(){//constructor
    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create a new drawing canvas and add it to
the
    // center of the JFrame.
    myCanvas = new MyCanvas();
    this.getContentPane().add(myCanvas);

    //This object must be visible before you can
get an
    // off-screen image. It must also be visible
before
    // you can compute the size of the canvas.
    setVisible(true);
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

    //Create an off-screen image and get a
graphics
    // context on it.
    osi = createImage(osiWidth,osiHeight);
    Graphics2D g2D = (Graphics2D)
(osi.getGraphics());

    //Draw some graphical objects on the off-
screen
    // image that represent underlying data

```

```

objects in
    // 2D space.
    drawOffScreen(g2D);

    //Cause the overridden paint method belonging
to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
//-----
-----//

    //The purpose of this method is to add some
Vector
    // objects and to cause visual manifestations of
the raw
    // Vector objects and the resultant Vector
objects to be
    // drawn onto an off-screen image.
    void drawOffScreen(Graphics2D g2D){

        //Translate the origin and draw a pair of
orthogonal
        // axes that intersect at the origin.
        setCoordinateFrame(g2D);

        //Define two vectors.
        GM2D04.Vector vecA = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(50, -100));
        GM2D04.Vector vecB = new GM2D04.Vector(
                                new
GM2D04.ColMatrix(-35, 110));

        //Define a third vector as the sum of the
first

```

```

// two vectors defined above.
GM2D04.Vector sumOf2 = vecA.add(vecB);

//Draw vecA in RED with its tail at the origin
g2D.setColor(Color.RED);
vecA.draw(g2D,new GM2D04.Point(
                                new
GM2D04.ColMatrix(0,0)));

//Draw vecB in GREEN with its tail at the head
of vecA
g2D.setColor(Color.GREEN);
vecB.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
vecA.getData(0),vecA.getData(1))));

//Draw sumOf2 in BLACK with its tail at the
origin.
// The head will coincide with the head of
vecB.
g2D.setColor(Color.BLACK);
sumOf2.draw(g2D,new GM2D04.Point(new
GM2D04.ColMatrix(
0.0,0.0)));

System.out.println(
                                "Red length = " +
vecA.getLength());
System.out.println(
                                "Green length = " +
vecB.getLength());
System.out.println(
                                "Black length = " +
sumOf2.getLength());
} //end drawOffScreen

```

```
//-----  
-----//
```

```
//This method is used to set the origin of the  
// off-screen image and to draw orthogonal axes  
that  
// intersect at the origin.  
private void setCoordinateFrame(Graphics2D g2D){
```

```
    //Translate the origin to a point near the  
lower-left
```

```
    // corner of the off-screen image.  
    g2D.translate(0.2*osiWidth,0.8*osiHeight);
```

```
    //Draw new X and Y-axes in default BLACK  
    g2D.drawLine(-(int)(0.2*osiWidth),0,  
                  (int)(0.8*osiWidth),0);
```

```
    g2D.drawLine(0,-(int)(0.8*osiHeight),  
                  0,(int)(0.2*osiHeight));
```

```
}//end setCoordinateFrame method
```

```
//=====
```

```
class MyCanvas extends Canvas{  
    //Override the paint() method. This method  
will be  
    // called when the JFrame and the Canvas  
appear on the  
    // screen or when the repaint method is called  
on the  
    // Canvas object.  
    //The purpose of this method is to display the
```

```

        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
//=====
=====//

```

Listing 24 . Source code for the program named VectorAdd04.

```

/*VectorAdd04.java
Copyright 2008, R.G.Baldwin
Revised 02/15/08

```

This program illustrates the addition of a Vector to a Point producing a new Point. Both points and the vector are drawn on the screen.

Tested using JDK 1.6 under WinXP.

```

*****
*****/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

class VectorAdd04{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    } //end main
} //end controlling class VectorAdd04
//=====
=====//

```

```

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 225;
    int vSize = 225;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to
the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.

```



```

        osi = createImage(osiWidth,osiHeight);
        Graphics2D g2D = (Graphics2D)
(osi.getGraphics());

        //Draw some graphical objects on the off-
screen
        // image.
        drawOffScreen(g2D);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    -----//

    //The purpose of this method is to add a Vector
object
    // to a Point object and to draw the result onto
an
    // off-screen image..
    void drawOffScreen(Graphics2D g2D){

        //Translate the origin on the off-screen
        // image and draw a pair of orthogonal axes on
it.
        setCoordinateFrame(g2D);

        //Define one point
        GM2D04.Point point = new GM2D04.Point(
                                new
GM2D04.ColMatrix(50,-25));

        //Define one vector.
        GM2D04.Vector vecA = new GM2D04.Vector(

```

```

new
GM2D04.ColMatrix(25, -50));

    //Add the Vector object to the Point object
producing
    // a new Point object
    GM2D04.Point newPoint =
point.addVectorToPoint(vecA);

    //Draw vecA in RED with its tail at the
original
    // point.
    g2D.setColor(Color.RED);
    vecA.draw(g2D, point);

    //Draw the original point in GREEN.
    g2D.setColor(Color.GREEN);
    point.draw(g2D);

    //Draw the new point in BLUE.
    g2D.setColor(Color.BLUE);
    newPoint.draw(g2D);

} //end drawOffScreen
//-----
-----//

    //This method is used to set the origin of the
    // off-screen image and to draw orthogonal axes
on the
    // off-screen image that intersect at the
origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Translate the origin to a point near the
lower-left
        // corner of the off-screen image.

```

```

        g2D.translate(0.2*osiWidth,0.8*osiHeight);

        //Draw new X and Y-axes in default BLACK
        g2D.drawLine(-(int)(0.2*osiWidth),0,
                     (int)(0.8*osiWidth),0);

        g2D.drawLine(0,-(int)(0.8*osiHeight),
                     0,(int)(0.2*osiHeight));

    }//end setCoordinateFrame method

//=====
====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the
        // screen or when the repaint method is called
on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
//=====
====//

```

Exercises

Exercise 1

Using Java and the game-math library named **GM2D04** , or using a different programming environment of your choice, write a program that creates twelve random values in the approximate range from -64 to +63. Use those values in pairs to define six mathematical vectors.

Draw the six vectors in a head-to-tail arrangement in alternating colors of green and blue with the tail of the first vector at the origin as shown in [Figure 8](#).

Compute and draw the sum of the six vectors in red with the tail of the sum vector at the origin.

Cause the origin of your reference frame to be at the center of your drawing and draw the axes for a Cartesian coordinate system in the reference frame in black.

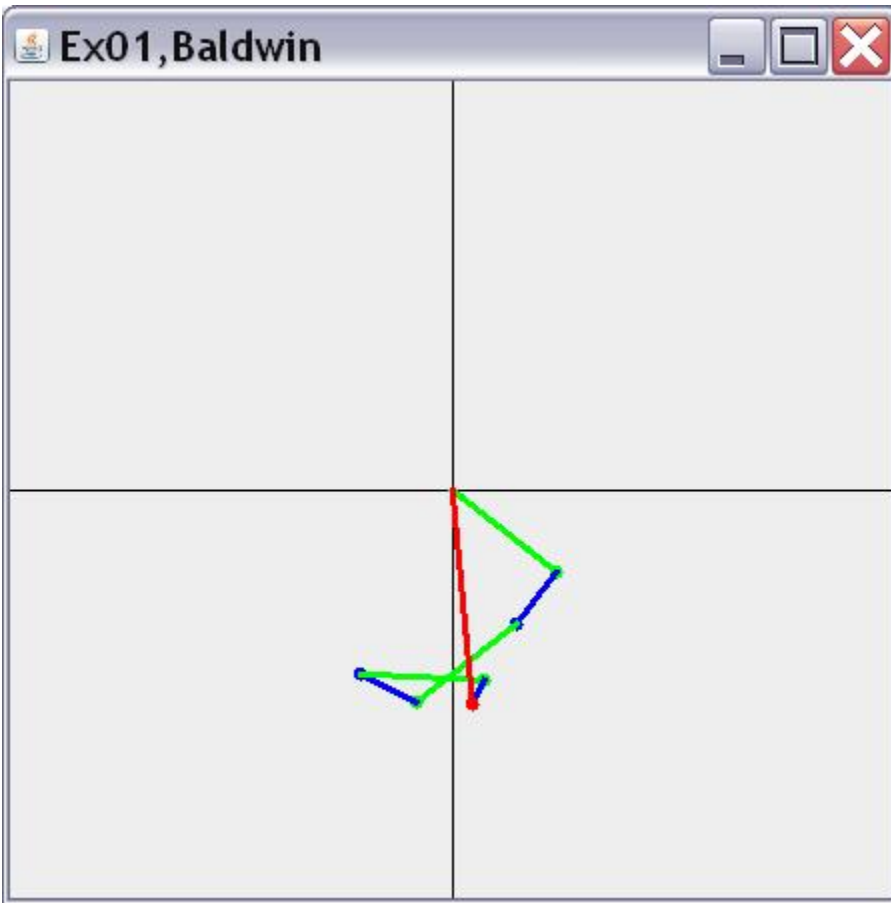
Cause the positive x direction to be to the right.

Cause the positive y direction be either up or down according to your choice.

Use a symbol of your choice to indicate the head of each vector.

Cause the program to display your name in some manner.

Figure 8 Screen output from Exercise 1.



Exercise 2

Using Java and the game-math library named **GM2D04** , or using a different programming environment of your choice, write a program that creates four random values in the approximate range from -128 to +127. Use those values in pairs to define two mathematical vectors.

Create a third vector as the sum of the first two vectors.

Using red, green, and blue, draw the three vectors in a way that illustrates the parallelogram rule for vector addition as shown in [Figure 9](#).

Cause the origin of your reference frame to be at the center of your drawing and draw the axes for a Cartesian coordinate system in the reference frame in black.

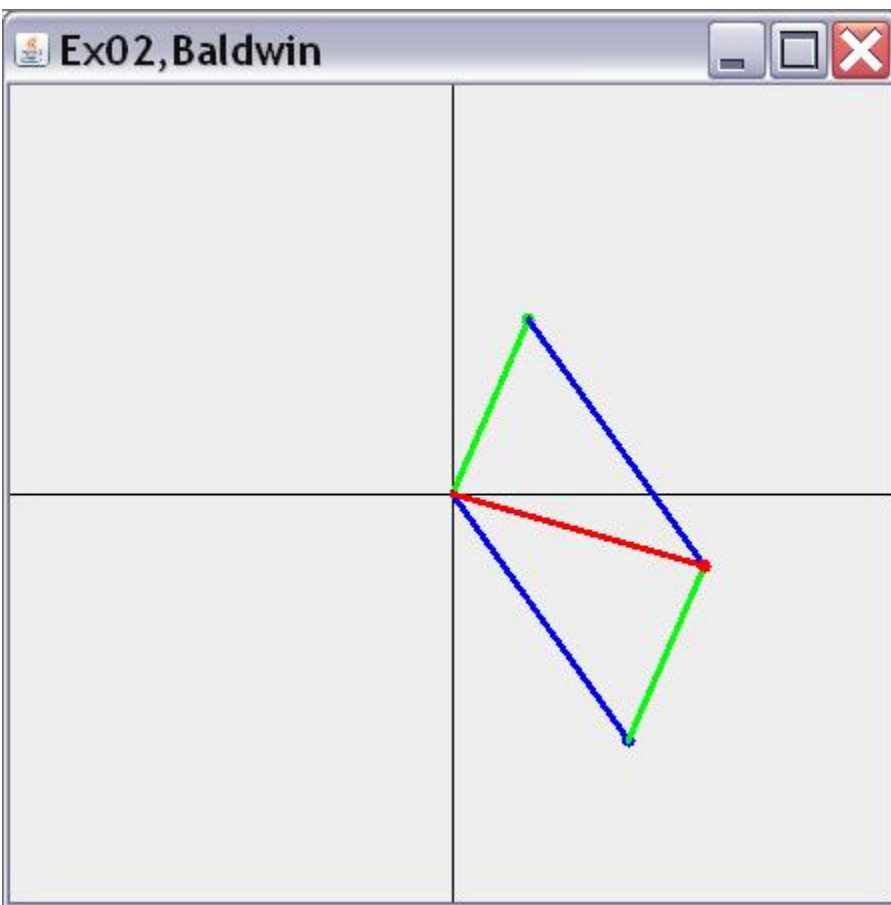
Cause the positive x direction to be to the right. Cause the positive y direction be either up or down according to your choice.

Use a symbol of your choice to indicate the head of each vector.

Cause the program to display your name in some manner.

Because of the random nature of the vectors, you may need to run your program more than once to open up the vector diagram and get a clear picture of the parallelogram rule for vector addition.

Figure 9 Screen output from Exercise 2.



-end-

Game0125r: Review

This module contains review questions and answers keyed to the module titled GAME 2302-0125: Vector Addition.

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module contains review questions and answers keyed to the module titled [GAME 2302-0125: Vector Addition](#).

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

True or False: The sum of two vectors is another vector.

[Answer 1](#)

Question 2

True or False: The length of a 2D vector is equal to the square root of the length of the hypotenuse of a right triangle formed by the lengths of the horizontal and vertical components of the vector.

[Answer 2](#)

Question 3

True or False: The sum of a vector and a point is another vector.

[Answer 3](#)

Question 4

True or False: A vector has only two properties: length and direction. It does not have a position property.

[Answer 4](#)

Question 5

True or False: When you draw a vector, you must draw it in a specific location.

[Answer 5](#)

Question 6

True or False: When you draw a vector, you can draw it anywhere in space, and one position is equally as valid as any other position.

[Answer 6](#)

Question 7

True or False: It's legal to draw a vector anywhere in space. Therefore, drawing the vector in one position doesn't have any advantage over drawing it in another position.

[Answer 7](#)

Question 8

True or False: If you add two or more vectors to produce a *sum* vector, and then draw the vectors that were included in the sum with the tail of one vector coinciding with the head of the previous vector, the head of the last vector that you draw will be positioned at some particular point in space.

If you then draw the vector that represents the sum vector with its tail coinciding with the tail of the first vector that you drew, its head will coincide with the head of the last vector that you drew.

[Answer 8](#)

Question 9

True or False: In order for the head-to-tail rule to apply, the coinciding tails must be drawn at the origin.

[Answer 9](#)

Question 10

True or False: The validity of the head-to-tail rule is limited to the sum of two vectors in two dimensions.

[Answer 10](#)

Question 11

True or False: A point has a location property. In fact that is the only property that a point does have. Because a point is simply a location in space, that location is always specified relative to some coordinate frame.

[Answer 11](#)

Question 12

True or False: The location of a point in world coordinates is independent of the values of the coordinates of the point with respect to changes in the coordinate frame.

[Answer 12](#)

Question 13

True or False: If an object is registered as an ActionListener on a button, the **doAction** method belonging to the object is called each time the user clicks the button.

[Answer 13](#)

Question 14

True or False: The result of adding vectors depends on the order in which you add the vectors.

[Answer 14](#)

Question 15

True or False: The length of the sum of two vectors is not necessarily equal to the sum of the lengths of the two vectors.

[Answer 15](#)

Question 16

True or False: The addition of a vector to a point produces a new point.

[Answer 16](#)

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 16

True

[Back to Question 16](#)

Answer 15

True. In fact, when the lengths of the vectors are the same and they are opposite in direction, the length of the sum of the two vectors is zero.

[Back to Question 15](#)

Answer 14

False. When you add vectors, the order in which you add the vectors doesn't matter. The result will be the same no matter the order in which you add

them.

[Back to Question 14](#)

Answer 13

False. If an object is registered as an ActionListener on a button, the **actionPerformed** method belonging to the object is called each time the user clicks the button.

[Back to Question 13](#)

Answer 12

False. In order to cause a point to remain in the same location relative to world coordinates, its values relative to the current coordinate frame must be modified each time the coordinate frame is modified.

[Back to Question 12](#)

Answer 11

True

[Back to Question 11](#)

Answer 10

False. The head-to-tail rule applies for the sum of any number of vectors in any number of dimensions (but it is difficult to draw in more than three dimensions).

[Back to Question 10](#)

Answer 9

False. It doesn't matter whether or not the coinciding tails are drawn at the origin. The head-to-tail rule still holds true.

[Back to Question 9](#)

Answer 8

True. This is a description of the head-to-tail rule.

[Back to Question 8](#)

Answer 7

False. While it's legal to draw a vector anywhere in space, certain positions may have advantages over other positions in some cases.

[Back to Question 7](#)

Answer 6

True

[Back to Question 6](#)

Answer 5

False. When you draw a vector, you can draw it anywhere in space, and one position is equally as valid as any other position.

[Back to Question 5](#)

Answer 4

True

[Back to Question 4](#)

Answer 3

False. The sum of a vector and a point is another *point* .

[Back to Question 3](#)

Answer 2

False. The length of a 2D vector is equal to the length of the hypotenuse of a right triangle formed by the lengths of the horizontal and vertical components of the vector.

[Back to Question 2](#)

Answer 1

True

[Back to Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0125r: Review: Vector Addition
- File: Game0125r.htm
- Published: 09/22/13
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0130: Putting the Game-Math Library to Work

Learn how to use the game-math library for translation and animation in two dimensions.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The game-math library named GM2D04](#)
 - [The program named VectorAdd05](#)
 - [The program named VectorAdd05a](#)
 - [The program named VectorAdd06](#)
 - [The program named StringArt01](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, you learned:

- how to add two or more vectors,
- about the head-to-tail rule in vector addition,
- about the vector addition parallelogram,
- about the relationship between the length of the sum of two or more vectors and the lengths of the individual vectors in the sum,
- how to add a vector to a point,
- how to get the length of a vector, and
- how to represent an object in different coordinate frames.

What you will learn

In this module we will put the game-math library to work. I will provide and explain four sample programs. The first program will teach you how to translate a geometric object in two dimensions. The second program will teach you how to accomplish the same thing but in a possibly more efficient manner. The third program will use the library to produce animation in two dimensions.

(A future module will produce translation and animation in three dimensions.)

The fourth program will teach you how to use methods of the game-math library to produce relatively complex drawings.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Translation of black hexagon to location of red hexagon.
- [Figure 2](#). Screen output for geometric object with 50 vertices.
- [Figure 3](#). Screen output without drawing the points at the vertices.
- [Figure 4](#). Starting point for the hexagon in VectorAdd06.
- [Figure 5](#). Possible ending point for the hexagon in VectorAdd06.
- [Figure 6](#). String art with 15 vertices and 7 loops.
- [Figure 7](#). String art with 25 vertices and 11 loops.
- [Figure 8](#). String art with 100 vertices and 100 loops.
- [Figure 9](#). Output from StringArt01 at startup.
- [Figure 10](#). Screen output from Exercise 1 at startup.

- [Figure 11](#). Screen output from Exercise 1 after clicking Replot button.

Listings

- [Listing 1](#). Instance variables in the class named GUI.
- [Listing 2](#). Abbreviated constructor for the GUI class.
- [Listing 3](#). Beginning of the drawOffScreen method.
- [Listing 4](#). To draw or not to draw the lines.
- [Listing 5](#). Change the drawing color to RED.
- [Listing 6](#). Translate the geometric object.
- [Listing 7](#). Draw the lines if drawLines is true.
- [Listing 8](#). The actionPerformed method.
- [Listing 9](#). Abbreviated listing of the drawOffScreen method.
- [Listing 10](#). The class named MyCanvas, the update method, and the paint method.
- [Listing 11](#). Abbreviated listing of actionPerformed method.
- [Listing 12](#). The inner Thread class named Animate.
- [Listing 13](#). Do the animated move.
- [Listing 14](#). Beginning of the drawOffScreen method in StringArt01.
- [Listing 15](#). Implement the algorithm that draws the lines.
- [Listing 16](#). Draw the lines.
- [Listing 17](#). Source code for the game-math library named GM2D04.
- [Listing 18](#). Source code for the sample program named VectorAdd05.
- [Listing 19](#). Source code for the program named VectorAdd05a.
- [Listing 20](#). Source code for the program named VectorAdd06.
- [Listing 21](#). Source code for the program named StringArt01.

Preview

In this module, I will present and explain four programs that use the game-math library named **GM2D04**. The purpose of the first program named **VectorAdd05** is to use the **addVectorToPoint** method of the **GM2D04.Point** class to translate a geometric object from one location in space to a different location in space.

The purpose of the program named **VectorAdd05a** is to accomplish the same translation operation, but in a possibly more efficient manner.

The purpose of the program named **VectorAdd06** is to teach you how to do rudimentary animation using the game-math library.

The purpose of the program named **StringArt01** is to teach you how to use methods of the game-math library to produce relatively complex drawings.

All of the programs are interactive in that they provide a GUI that allows the user to modify certain aspects of the behavior of the program.

I will also provide an exercise for you to complete on your own at the end of the module. The exercise will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

The game-math library named GM2D04

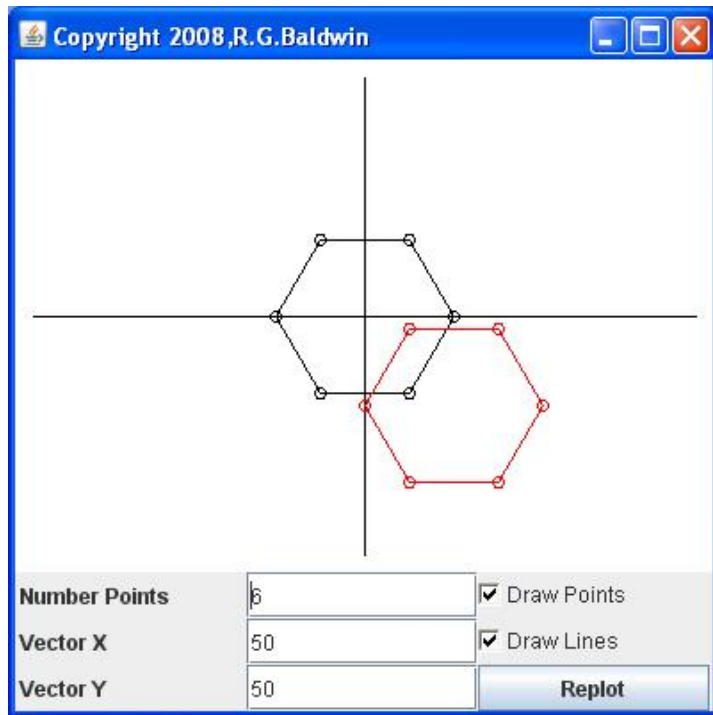
The game-math library that we are developing in this collection of modules has undergone several updates since its inception in the first module. The version of the library that existed at the end of the previous module was named **GM2D04** . No revisions have been made to the library since that module and we won't be making any revisions during this module.

The source code for the library named **GM2D04** is provided for your convenience in [Listing 17](#) near the end of the module. Documentation for the library was provided in the earlier module titled [GAME 2302-0125: Vector Addition](#) .

The program named VectorAdd05

The main purpose of this program is to use the **addVectorToPoint** method of the **GM2D04.Point** class in the game-math library to translate a geometric object from one location in space to a different location in space. This is illustrated in [Figure 1](#), which shows one hexagon (*shown in black*) having been translated by 50 units in both the x and y directions and colored red at the new location.

Figure 1 Translation of black hexagon to location of red hexagon.



Various other game-math library methods are also used

Along the way, the program uses various other methods of the classes in the game-math library named **GM2D04** to accomplish its purpose.

The program initially constructs and draws a black hexagon centered on the origin as shown in [Figure 1](#). The six points that define the vertices of the hexagon lie on a circle with a radius of 50 units. The points at the vertices and the lines that connect the points are drawn. In addition, the program causes the hexagon to be translated by 50 units in the positive X direction and 50 units in the positive Y direction. The translated hexagon is drawn in red. The original black hexagon is not erased when the translated version is drawn in red.

A graphical user interface (GUI)

A GUI is provided that allows the user to specify the following items and click a **Replot** button to cause the drawing to change:

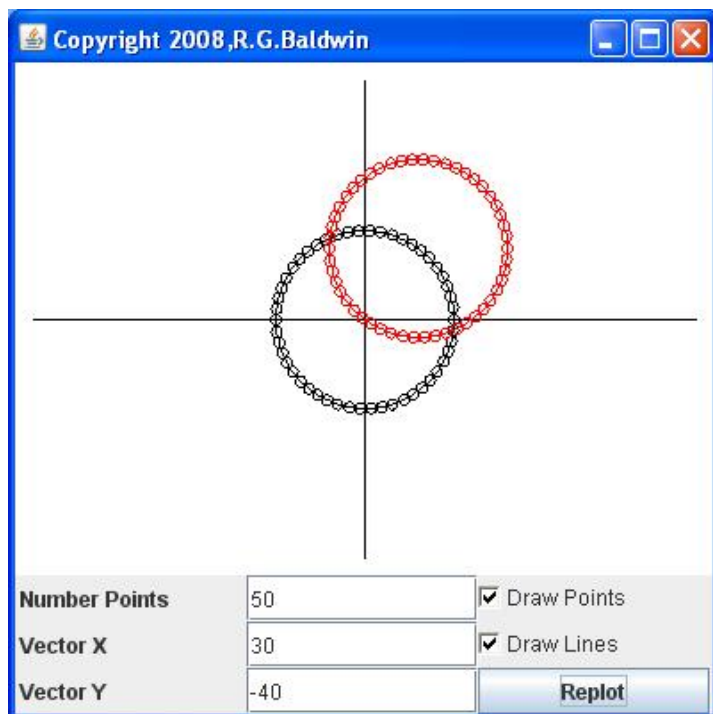
- Number of points to define the geometric object.
- X-component of the displacement vector.
- Y-component of the displacement vector.
- A checkbox to specify whether points are to be drawn.
- A checkbox to specify whether lines are to be drawn.

These user-input features are shown at the bottom of [Figure 1](#).

Changing the number of points

Changing the number of points causes the number of vertices that describe the geometric object to change. For a large number of points, the geometric object becomes a circle as shown in [Figure 2](#).

Figure 2 Screen output for geometric object with 50 vertices.



For only three points, the geometric object would become a triangle. For four points, it would become a rectangle. For two points, it would become a line, etc.

Translation

Changing the components of the displacement vector causes the geometric object to be translated to a different location before being drawn in red. In addition to increasing the number of vertices, [Figure 2](#) also shows the result of translating by 30 units along the x-axis and -40 units along the y-axis.

Note: The positive y-axis:

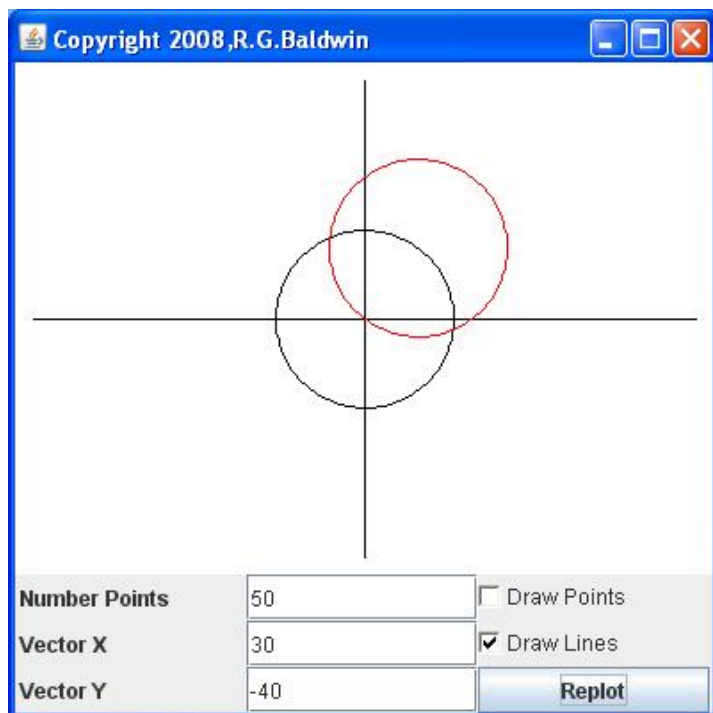
The positive direction for the y-axis is down in the figures in this module. This is the default for Java graphics. I will resolve that issue and cause the positive direction for

the y-axis to be up instead of down in a future module.

Checking and un-checking the checkboxes

[Figure 3](#) shows the result of un-checking one of the checkboxes to prevent the points that define the vertices from being drawn. In this case, only the lines that connect the vertices were drawn, resulting in the two circles shown.

Figure 3 Screen output without drawing the points at the vertices.



Similarly, the GUI can be used to cause only the points that define the vertices to be drawn without the connecting lines.

Will explain the code in fragments

I will explain the code in this program in fragments. A complete listing of the program is provided in [Listing 18](#) for your convenience.

Much of the code in this module is very similar to code that I have explained in earlier modules, so I won't bore you by repeating those explanations. Rather, I will concentrate on code that is new and different in this module.

Beginning of the class named GUI

The beginning of the class named **GUI** is shown in [Listing 1](#).

Listing 1 . Instance variables in the class named GUI.

```
class GUI extends JFrame implements ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//Off-screen graphics context.

    //The following two variables are used to establish
    the
    // location of the origin.
    double xAxisOffset;
    double yAxisOffset;

    int numberPoints = 6;//Can be modified by the user.
    JTextField numberPointsField; //User input field.
    //The components of the following displacement
    vector
    // can be modified by the user.
    GM2D04.Vector vector =
        new GM2D04.Vector(new
    GM2D04.ColMatrix(50,50));
    JTextField vectorX;//User input field.
    JTextField vectorY;//User input field.

    //The following variables are used to determine
    whether
    // to draw the points and/or the lines.
```


Listing 1 . Instance variables in the class named GUI.

```
boolean drawPoints = true;
boolean drawLines = true;
Checkbox drawPointsBox;//User input field
Checkbox drawLinesBox;//User input field.

//The following variables are used to refer to
array
// objects containing the points that define the
// vertices of the geometric object.
GM2D04.Point[] points;
GM2D04.Point[] newPoints;
```

The code in [Listing 1](#) declares a large number of instance variables, initializing some of them. Those variables shouldn't require an explanation beyond the embedded comments. I show them here solely to make it easy for you to refer to them later when I discuss them.

Extends JFrame and implements ActionListener

This class extends the **JFrame** class and implements the **ActionListener** interface. As you will see later, implementing the **ActionListener** interface requires that the class contains a concrete definition of the **actionPerformed** method. It also makes an object of the **GUI** class eligible for being registered as a listener object on the **Replot** button shown in [Figure 1](#).

Abbreviated constructor for the GUI class

An Abbreviated listing of the constructor for the **GUI** class is shown in [Listing 2](#). Much of the code was deleted from [Listing 2](#) for brevity. You can view the code that was deleted in [Listing 18](#).

Listing 2 . Abbreviated constructor for the GUI class.

Listing 2 . Abbreviated constructor for the GUI class.

```
GUI(){//constructor
    //Instantiate the array objects that will be used
to
    // store the points that define the vertices of
the
    // geometric object.
    points = new GM2D04.Point[numberPoints];
    newPoints = new GM2D04.Point[numberPoints];

//Code that creates the user interface was deleted
// for brevity.

//Code dealing with the canvas and the off-screen
image
// was deleted for brevity.

    //Erase the off-screen image and draw the axes
setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,true);

    //Create the Point objects that define the
geometric
    // object and manipulate them to produce the
desired
    // results.
    drawOffScreen(g2D);

    //Register this object as an action listener on
the
    // button.
    button.addActionListener(this);

    //Cause the overridden paint method belonging to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
```

Arrays for storage of vertices

The constructor begins by instantiating two array objects that will be used to store references to the **GM2D04.Point** objects that define the vertices of the geometric object.

Set the coordinate frame

Further down in [Listing 2](#), the constructor calls the method named **setCoordinateFrame** to erase the off-screen image, draw orthogonal axes on it, and translate the origin to a new location. The code in the **setCoordinateFrame** method is straightforward and shouldn't require an explanation beyond the embedded comments. You can view that code in [Listing 18](#).

The fourth parameter to the **setCoordinateFrame** method is used to determine whether or not to translate the origin by the amount given by the second and third parameters. If true, the origin is translated. If false, the origin is not translated.

The method named drawOffScreen

After setting the coordinate frame, [Listing 2](#) calls the method named **drawOffScreen**. I will discuss that method in some detail shortly.

Register an ActionListener object

Next, the code in [Listing 2](#) registers the object of the **GUI** class as an action listener on the **Replot** button shown in [Figure 1](#). This causes the method named **actionPerformed** to be executed whenever the user clicks the **Replot** button. I will also discuss the **actionPerformed** method shortly.

Repaint the canvas

Finally, [Listing 2](#) calls the **repaint** method to repaint the canvas. If you have studied the previous modules in this collection, you should know what this does and an explanation should not be necessary.

Beginning of the drawOffScreen method

[Listing 3](#) shows the beginning of the method named **drawOffScreen**. The purpose of this method is to create the **GM2D04.Point** objects that define the vertices of the geometric object and to manipulate those points to produce the desired results.

Listing 3 . Beginning of the drawOffScreen method.

```
void drawOffScreen(Graphics2D g2D){
    //Erase the off-screen image and draw new axes,
but
    // don't change the coordinate frame.

    setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,false);

    //Create a set of Point objects that specify
    // locations on the circumference of a circle and
    // save references to the Point objects in an
array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM2D04.Point(
            new GM2D04.ColMatrix(

50*Math.cos((cnt*360/numberPoints)

*Math.PI/180),

50*Math.sin((cnt*360/numberPoints)

*Math.PI/180)));
        if(drawPoints){//Draw points if true
            points[cnt].draw(g2D);
        }//end if
    }//end for loop
```

Erase the image and draw the axes

[Listing 3](#) begins by calling the **setCoordinateFrame** method to erase the off-screen image and draw orthogonal axes, intersecting at the origin on the erased image. Note however that unlike the call to the **setCoordinateFrame** method that was made in the constructor in [Listing 2](#), the call to the method in [Listing 3](#) passes **false** as the fourth parameter, thereby preventing the location of the origin from being modified.

In other words, in this case, the method is being called simply to erase the off-screen image and to draw axes on the clean off-screen image.

(In hindsight, it may have been better to break this method into two separate methods; one to translate the origin and a second to erase the off-screen image and draw the axes.)

Create the points that define the vertices

Then [Listing 3](#) executes a **for** loop that instantiates the set of mathematical **GM2D04.Point** objects that define the vertices of the geometric object and saves references to those objects in the array object that was instantiated in [Listing 2](#). Note that during the first pass through the constructor and the method named **drawOffScreen**, the **length** of the array object and the number of points instantiated are both specified by the initial value (6) of the instance variable named **numberPoints** (see [Listing 1](#)).

Some knowledge of trigonometry is required

I told you in an earlier module *"I will assume that you either already have, or can gain the required skills in geometry and trigonometry on your own."* I included a module titled [GAME 2302-0320 Brief Trigonometry Tutorial](#) in this collection to assist you in that effort.

In order to understand the code in the **for** loop in [Listing 3](#), you must have at least a rudimentary knowledge of trigonometry.

For now, suffice it to say that this code will instantiate a set of **GM2D04.Point** objects equally spaced around the circumference of a circle with a radius of 50 units centered on the origin.

(When rendered on the off-screen image, these units will be translated to pixels.)

For the initial value of six points, the first point will be located at an angle of zero degrees relative to the horizontal, and each of the remaining five points will be located on the circumference of the circle at an angle that is an even multiple of $360/6$ or 60 degrees.

To draw or not to draw the points

Recall that a *point* has no width, no height, and no depth and is therefore not visible to the human eye. However, when you call the **draw** method on an object of the **GM2D04.Point** class, a small circle is drawn around the location of the point marking that location for human consumption.

An **if** statement embedded in the **for** loop in [Listing 3](#) tests the value of the **boolean** instance variable named **drawPoints** (see [Listing 1](#), which initializes the value to *true*) to determine whether or not to draw the circle marking the location of each point as it is instantiated and saved in the array object. If **true**, the circle is drawn as shown in [Figure 1](#). If **false**, the circle is not drawn as shown in [Figure 3](#). As you will see later, the user can modify the value of the variable named **drawPoints** using one of the checkboxes and the **Replot** button in [Figure 1](#).

Note: The default drawing color:

The default drawing color is BLACK. When the points are drawn the first time the **drawOffScreen** method is called, they will be drawn in BLACK, which is a public static final variable in the **Color** class.

To draw or not to draw the lines

[Listing 4](#) tests the value of the instance variable named **drawLines** to determine whether or not to draw lines connecting each of the points. This variable is also initialized to **true** in [Listing 1](#), and its value can be modified by the user later using one of the checkboxes and the **Replot** button shown in [Figure 1](#).

Listing 4 . To draw or not to draw the lines.

Listing 4 . To draw or not to draw the lines.

```
GM2D04.Line line;
if(drawLines){//Instantiate and draw lines if
true.
    for(int cnt = 0;cnt < numberPoints-1;cnt++){
        //Begin by drawing all of the lines but one.
        line = new
GM2D04.Line(points[cnt],points[cnt+1]);
        line.draw(g2D);
    }//end for loop
    //Draw the remaining line required to close the
    // polygon.
    line = new GM2D04.Line(points[numberPoints-1],
        points[0]);
    line.draw(g2D);
} //end if
```

Slightly more complicated

Drawing the lines is only slightly more complicated than drawing the points. A **for** loop is used in [Listing 4](#) to draw lines connecting successive pairs of points whose references were stored in the array named **points** in [Listing 3](#). This takes care of all of the required lines but one. Following the **for** loop, one additional statement is executed to draw a line connecting the points whose references are stored in the first and the last elements in the array.

*(I will show you another way to accomplish this wraparound in the program named **StringArt01** later in this module.)*

No need to save the GM2D04.Line objects

Note that the **GM2D04.Line** object that is used to draw each connecting line has no value in this program after the line is drawn on the off-screen image. Therefore, there is no need to consume memory by saving a large number of such objects. A single reference variable of the class **GM2D04.Line** is used to refer to all the objects of the **GM2D04.Line** class that are used to draw the connecting lines. As each new object of that class is instantiated, the object previously referred to by the reference variable becomes *eligible for garbage collection* .

(Go to Google and search for the following keywords to learn more about this topic: baldwin java eligible garbage collection.)

Change the drawing color to RED

After the BLACK geometric object shown in [Figure 1](#) has been drawn on the off-screen image, it is time to change the drawing color to RED in preparation for translating the object and drawing the translated version of the object. This is accomplished by the single statement in [Listing 5](#).

Listing 5 . Change the drawing color to RED.

```
g2D.setColor(Color.RED); //Change drawing color to RED.
```

Translate the geometric object

That brings us to the most important block of code in the entire program, insofar as achieving the main learning objective of the program is concerned.

Listing 6 . Translate the geometric object.

Listing 6 . Translate the geometric object.

```
for(int cnt = 0;cnt < numberPoints;cnt++){
    newPoints[cnt] =
points[cnt].addVectorToPoint(vector);
    if(drawPoints){//Draw points if true.
        newPoints[cnt].draw(g2D);
    }//end if
} //end for loop
```

[Listing 6](#) uses a call to the **addVectorToPoint** method of the **GM2D04.Point** class embedded in a **for** loop to *translate* each of the vertices (*points*) that define the original geometric object to a second set of vertices that define a geometric object having the same shape in a different location.

*(The initial **vector** that was used to translate the points in [Listing 6](#) was instantiated in [Listing 1](#) with values of 50,50. That vector may be modified later using the values in the fields named **Vector X** and **Vector Y** in the GUI shown in [Figure 1](#) when the user clicks the **Replot** button.)*

The new vertices were saved in a different array object referred to by the reference variable named **newPoints** . It is important to note that in this case, the original geometric object was not moved. Rather, it was replicated in a new location with the new location being defined by a displacement vector added to the value of each original point.

Make it appear to move

In many cases in game programming, you will want to make it appear that the object has actually moved instead of being replicated. That appearance could be achieved by saving the reference to the new **GM2D04.Point** object back into the same array, thereby replacing the reference that was previously there. Make no mistake about it, however, when using this approach, the translated geometric object is a different object, defined by a new set of **GM2D04.Point** objects that define the vertices of the geometric object in the new location.

A different approach

A **different approach** would be to call the **setData** method of the **GM2D04.Point** class to modify the coordinate values stored in the existing object that define the location of the point. In that case, it would not be necessary to instantiate a new object, and I suppose it could be argued that such an approach would actually *move* each vertex, thereby *moving* the geometric object. If execution speed is an important factor (*particularly in animation code*) it would probably be useful to run some benchmarks to determine which approach would be best. (*I will show you how to implement this approach in the program named **VectorAdd05a** later in this module.*)

Drawing the points

Once again, an **if** statement is embedded in the code in [Listing 6](#) to determine whether or not to draw the new points on the off-screen image as the new **GM2D04.Point** objects are instantiated and saved in the array. If the points are drawn, they will be drawn in RED due to the code in [Listing 5](#).

Drawing the lines

The code in [Listing 7](#) is essentially the same as the code in [Listing 4](#). This code tests the variable named **drawLines** to determine whether or not to draw lines connecting the new points in the current drawing color (*RED*) .

Listing 7 . Draw the lines if drawLines is true.

Listing 7 . Draw the lines if drawLines is true.

```
        if(drawLines){//Instantiate and draw lines if
true.
            for(int cnt = 0;cnt < numberPoints-1;cnt++){
                line = new GM2D04.Line(newPoints[cnt],
newPoints[cnt+1]);
                line.draw(g2D);
            }//end for loop
            //Draw the remaining line required to close the
            // polygon.
            line = new GM2D04.Line(newPoints[numberPoints-
1],
newPoints[0]);
            line.draw(g2D);
        }//end if

    }//end drawOffScreen
```

End of the method

[Listing 7](#) also signals the end of the method named **drawOffScreen** .

The actionPerformed method

As mentioned earlier, the **actionPerformed** method, shown in [Listing 8](#), is called to respond to each click on the **Replot** button shown in [Figure 1](#).

Listing 8 . The actionPerformed method.

```

Listing B1. Conditional Performed method
//Get user input values and use them to modify
several
// values that control the translation and the
// drawing.
numberPoints = Integer.parseInt(
numberPointsField.getText());
vector.setData(
0, Double.parseDouble(vectorX.getText()));
vector.setData(
1, Double.parseDouble(vectorY.getText()));

if(drawPointsBox.getState()){
    drawPoints = true;
}else{
    drawPoints = false;
} //end else

if(drawLinesBox.getState()){
    drawLines = true;
}else{
    drawLines = false;
} //end else

//Instantiate two new array objects with a length
// that matches the new value for numberPoints.
points = new GM2D04.Point[numberPoints];
newPoints = new GM2D04.Point[numberPoints];

//Draw a new off-screen image based on user
inputs.
drawOffScreen(g2D);
myCanvas.repaint();//Copy off-screen image to
canvas.
} //end actionPerformed

```

Behavior of the actionPerformed method

Basically this method:

- Gets input values from the user input components shown at the bottom of [Figure 1](#).
- Uses the input values to set the values of certain variables.
- Instantiates new array objects of the correct length to contain the points that define the vertices of the geometric objects.
- Calls the **drawOffScreen** method to generate the new point and line objects and display them on the canvas.

End of the program discussion

That ends the discussion of the program named **VectorAdd05** .

The program named VectorAdd05a

I promised you earlier that I would explain a [different approach](#) to modifying the vertices of the geometric object in order to translate it. That approach is used in the program named **VVectorAdd05a** , which you will find in [Listing 19](#).

The only significant change to this program relative to the program named **VectorAdd05** is shown in [Listing 9](#).

Listing 9 . Abbreviated listing of the drawOffScreen method.

Listing 9 . Abbreviated listing of the drawOffScreen method.

```
void drawOffScreen(Graphics2D g2D){
//Code deleted for brevity.

    g2D.setColor(Color.RED); //Change drawing color to
    RED.

    //Translate the geometric object and save the
    points
    // that define the translated object in the same
    // array object.
    for(int cnt = 0; cnt < numberPoints; cnt++){
        points[cnt].setData(
            0, points[cnt].getData(0) +
vector.getData(0));
        points[cnt].setData(
            1, points[cnt].getData(1) +
vector.getData(1));

        if(drawPoints){ //Draw points if true.
            points[cnt].draw(g2D);
        } //end if
    } //end for loop

//Code deleted for brevity
} //end drawOffScreen
```

Compare this code with an earlier listing

You should compare the code in [Listing 9](#) with the code in [Listing 6](#). The code in [Listing 9](#) does not call the **addVectorToPoint** method to instantiate a new set of **GM2D04.Point** objects. Instead, it uses lower-level methods from the game-math library to modify the x and y attribute values that define the locations of the existing **GM2D04.Point** objects.

As you can see, the code in [Listing 9](#) is somewhat more complex than the code in [Listing 6](#). However, the code in [Listing 9](#) possibly entails lower overhead at runtime. First, it doesn't call the **addVectorToPoint** method that hides the programming

details behind a single method call. Second, the code in [Listing 9](#) doesn't instantiate new objects of the **GM2D04.Point** class, as is the case in [Listing 6](#).

A middle ground

A middle ground that is not illustrated here might be to use a method like the **addVectorToPoint** method to hide the details, but to have that method change the x and y attribute values in the existing points instead of instantiating new objects. None of the three approaches is either right or wrong. There are pros and cons to all three approaches. This is indicative of the kinds of decisions that must be made by developers who design class libraries containing classes and methods to accomplish common tasks.

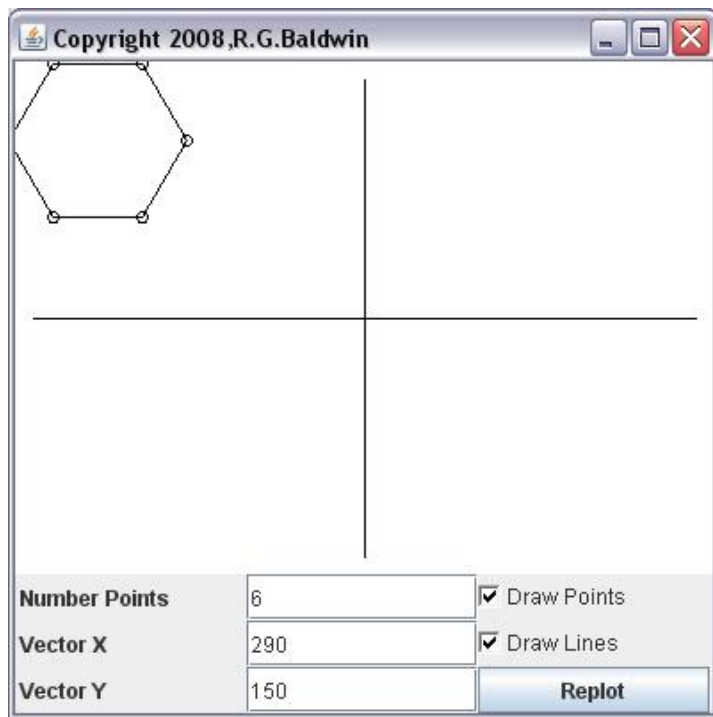
End of the program discussion

That ends the discussion of the program named **VectorAdd05a** .

The program named VectorAdd06

This is an update to the program named **VectorAdd05** . The behavior of this program is similar to the earlier program except that instead of displaying a static view of the translated geometric object when the **Replot** button is clicked, this program animates the geometric object causing it to move from its original location shown in [Figure 4](#) to a new location in 100 incremental steps along a straight line path.

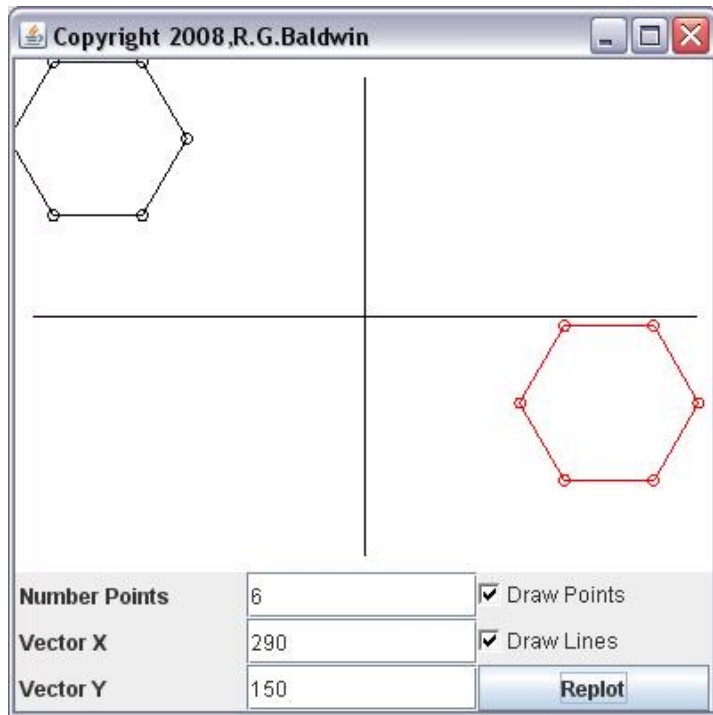
Figure 4 Starting point for the hexagon in VectorAdd06.



The final position

The final position for the translated hexagon depends on the user input values for the vector. [Figure 5](#) shows one possible ending point for the translated hexagon.

Figure 5 Possible ending point for the hexagon in VectorAdd06.



The animation loop

The animation loop sleeps for ten milliseconds during each of the 100 iterations. Therefore, approximately one second (*1000 milliseconds and possibly more*) is required for the object to make the trip from its initial location to its new location. Once it reaches the new location, the program is ready for the user to change input values and click the **Replot** button again.

(Each time you click the Replot button, the object can be seen to move from the initial location, not the current location, to the new location.)

As with the program named **VectorAdd05**, this program uses the **addVectorToPoint** method of the **GM2D04.Point** class to translate a geometric object from one location in space to a different location in space. In this program, however, the translation is performed in 100 incremental steps to produce an animation. As shown in [Figure 5](#), the animated geometric object is drawn in red to make it visually distinct from the original object. The original object is not erased from the display.

The polygon

The program initially constructs and draws a black polygon in the upper-left corner of the canvas as shown in [Figure 4](#). (*The polygon is a hexagon in [Figure 4](#).*) The six points that define the vertices of the hexagon lie on a circle with a radius of 50 units. The points at the vertices and the lines that connect the points are drawn initially.

Also as shown in [Figure 4](#), a GUI is provided that allows the user to specify the following items and click a **Replot** button to cause the animation to begin:

- The number of points that define the geometric object (*polygon*) .
- X-component of the displacement vector.
- Y-component of the displacement vector.
- A checkbox to specify whether points are to be drawn.
- A checkbox to specify whether lines are to be drawn.

As in the previous program, changing the number of points causes the number of vertices that describe the geometric object to change, allowing you to create triangles, rectangles, pentagons, hexagons, circles, etc.

Changing the components of the displacement vector causes the geometric object to be translated to a different location. Checking and unchecking the checkboxes causes the points and/or the lines to either be drawn or not drawn.

Animation performance

On the computer that I am now using, which is not a new one (*September 2012*), the animation becomes slightly jerky at about 3500 points when both the points and the lines are drawn.

Will explain in fragments

I will explain this program in fragments and will avoid repeating explanations that I have previously given. A complete listing of this program is provided in [Listing 20](#) near the end of the module.

Much of the code in this program is very similar to code that I explained earlier. The thing that is new about this program is the animation aspect of the program. Therefore, I will concentrate mainly on the animation code in my explanation of the program.

The class named MyCanvas, the update method, and the paint method

Before getting into the details of the program, I want to explain the inner workings of the class named **MyCanvas** that I defined for this program. This class, which is an inner class of the **GUI** class, is shown in its entirety in [Listing 10](#).

Listing 10 . The class named MyCanvas, the update method, and the paint method.

```
class MyCanvas extends Canvas{
    public void update(Graphics g){
        paint(g); //Call the overridden paint method.
    } //end overridden update()
    //-----
---//

    public void paint(Graphics g){
        g.drawImage(osi, 0, 0, this);
    } //end overridden paint()

} //end inner class MyCanvas
```

Different from previous version

If you compare this class with the class named **MyCanvas** in the earlier program named **VectorAdd05** (see [Listing 18](#)) , you will see that this version of the class is considerably different from the previous version.

The previous version simply overrode the method named **paint** to cause the off-screen image to be copied to the canvas. That is a perfectly good approach for programs where the image that is displayed on the canvas changes only occasionally. However, that is not a good approach for an animation program where the image displayed on the canvas changes frequently.

Calling the update method

Recall that in Java, when we want to display an image on the computer screen, we don't call the overridden **paint** method directly. Rather, we call the method named **repaint** .

Among other tasks, the **repaint** method acts as a traffic cop to decide which application will be allowed to draw on the screen next, but that isn't the main thrust of our interest here. The main thrust of our interest here is that when we call the **repaint** method, this eventually results in a call to a method named **update** . By default, the **update** method paints a white background on the canvas erasing anything that was

previously there. Then the **update** method calls our overridden **paint** method to draw whatever it is that we want to draw on the canvas.

Can cause undesirable flashing

Normally this is an adequate approach. However, for an animation program, the repetitive painting of a white background on the canvas can cause an undesirable level of flashing in the visual output of the program. In this case, there is no need to paint a white background on the canvas before copying the off-screen image on the canvas because that image will completely fill the canvas with the new image. Therefore, allowing the canvas to be painted white each time the off-screen image is copied to the canvas is unnecessary and does cause flashing on my computer.

Overriding the update method

[Listing 10](#) overrides the **update** method to eliminate the painting of a white background on the canvas. In [Listing 10](#), the update method simply calls the overridden **paint** method to copy the off-screen image onto the canvas. On my computer, this results in a much more pleasing animation output.

The actionPerformed method

The next code fragment that I will explain is the **actionPerformed** method, which is called in response to each click on the **Replot** button shown in [Figure 4](#) and [Figure 5](#).

[Listing 11](#) shows an abbreviated listing of the **actionPerformed** method.

Listing 11 . Abbreviated listing of actionPerformed method.

```
public void
```

Most of the code in this method is the same as code that I explained in conjunction with [Listing 8](#), so I won't repeat that explanation here.

(I deleted that code for brevity in [Listing 11](#). You can view the method in its entirety in [Listing 20](#).)

The code in [Listing 11](#) picks up with a statement that sets the value of an instance variable named **animation** to true. This variable is used simply to prevent the animation from starting until the first time the user clicks the **Replot** button.

Spawn an animation thread

This is where this program really departs from the previous programs. [Listing 11](#) instantiates an object of an inner **Thread** class named **Animate** and saves that object's reference in a local variable named **animate** . Then [Listing 11](#) uses that variable to call the **start** method on the **Animate** object.

I won't attempt to go into all of the technical details involved in this operation. Suffice it to say that this eventually causes a method named **run** belonging to the **Animate** object to be executed.

The Animate class and the run method

The **Animate** class and the **run** method begin in [Listing 12](#).

Listing 12 . The inner Thread class named Animate.

```
class Animate extends Thread{
    public void run(){
        //Compute the incremental distances that the
        // geometric object will move during each
iteration.
        double xInc = vector.getData(0)/100;
        double yInc = vector.getData(1)/100;
```

The **run** method begins by computing incremental x and y displacement values that will be required to move the geometric object from its initial position as shown in [Figure 4](#) to its final position as shown in [Figure 5](#) in 100 equal steps.

Do the animated move

Then [Listing 13](#) executes 100 iterations of a **for** loop to cause the geometric object to actually move through those 100 incremental steps and to be drawn on the screen once during each incremental step.

Listing 13 . Do the animated move.

```
for(int cnt = 0;cnt < 100;cnt++){
    vector.setData(0,cnt*xInc);
    vector.setData(1,cnt*yInc);

    //Draw a new off-screen image based on the
    // incremental displacement vector and other
user    // inputs.
        drawOffScreen(g2D);
        //Copy off-screen image to canvas.
        myCanvas.repaint();

        //Sleep for ten milliseconds.
        try{
            Thread.currentThread().sleep(10);
        }catch(InterruptedException ex){
            ex.printStackTrace();
        }//end catch
    }//end for loop
}//end run
}//end inner class named Animate
```

During each iteration of the for loop...

During each iteration of the **for** loop in [Listing 13](#), an incremental displacement vector is created with X and Y component values that are equal to 1/100 of the user-specified displacement vector multiplied by the loop counter value. The incremental displacement vector is used by the code in the **drawOffScreen** method to translate the geometric object to a new location on the off-screen image. Then the **repaint**

method is called on the canvas to cause the off-screen image to be copied onto the canvas as described in conjunction with [Listing 10](#) earlier.

*(Note that the object is translated from its original location, not its current location, during each iteration of the **for** loop. A different approach would be to save the current location and translate it from the current location during each iteration of the **for** loop.)*

At the end of each iteration of the **for** loop, the animation thread *sleeps* for ten milliseconds before starting the next iteration.

The bottom line is that this code causes the geometric object to move in incremental steps from its **initial location** to a new location based on a displacement vector specified by the user. In other words, the geometric object reaches its final location in a straight-line animated manner, taking 100 steps to get there.

The drawOffScreen method

The **drawOffScreen** method that is called by the **Animation** thread in [Listing 13](#) is very similar to the method having the same name that I explained in conjunction with [Listing 3](#) through [Listing 7](#). Therefore, I won't repeat that explanation here.

The big difference in the use of that method in the two programs is that in the earlier program, the method was called only once to translate the geometric object according to the values in the displacement vector in one giant step. In this program, the values in the displacement vector are divided by 100 and the **drawOffScreen** method is called 100 times with ten-millisecond pauses between each call to the method. Each time the **drawOffScreen** method is called, the geometric object ends up closer to its final position and each of the 100 intermediate positions are displayed on the canvas for ten milliseconds.

End of discussion

That ends the discussion of the program named **VectorAdd06**.

The program named StringArt01

You may be old enough to remember when people who liked to do handicraft projects created pictures using strings and nails. If not, I'm sure that if you Google the topic of *string art*, you will find more about the topic than you ever wanted to know.

This program was inspired by some programs that I saw at the [Son of String Art](#) page. I will be the first to admit that the output from this program isn't nearly as elaborate or interesting as the output from some of the projects at that web site. Those programs are truly impressive, particularly considering that they were written using a programming language that was *"designed to help young people (ages 8 and up) develop 21st century learning skills."* (See [About Scratch](#).)

In future modules, however, I will show you how to improve this string art program including the addition of animated translation and rotation of the string-art images in 3D.

Behavior of the program

This program produces a 2D string art image by drawing lines to connect various points that are equally spaced around the circumference of a circle. At startup, the program constructs and draws a multi-colored hexagon centered on the origin as shown in [Figure 9](#). The six points that define the vertices of the hexagon lie on a circle with a radius of 100 units. The points at the vertices are not drawn, but the lines that connect the vertices are drawn.

A GUI is provided that allows the user to specify the following items and click a **Replot** button to cause the drawing to change:

- Number Points
- Number Loops

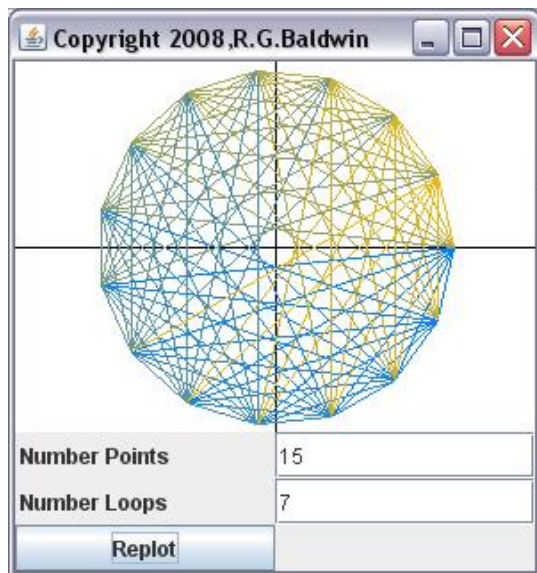
As in earlier programs in this module, changing the number of points causes the number of vertices that describe the geometric object to change. Changing the number of loops causes the number of lines that are drawn to connect the vertices to change. For a *Loop* value of 1, each vertex is connected to the one next to it. For a value of 2, additional lines are drawn connecting every other vertex. For a value of 3, additional lines are drawn connecting every third vertex, etc.

Making the number of points and loops large produces some interesting patterns.

Some output from the program

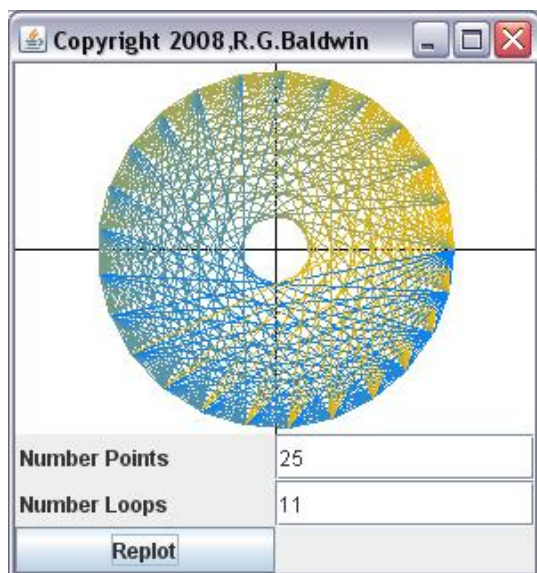
Before getting into the technical details of the program, I am going to show you some sample screen output from the program. [Figure 6](#) shows the result of drawing lines to connect the points in the pattern described above among fifteen points that are equally spaced around the circumference of the circle.

Figure 6 String art with 15 vertices and 7 loops.



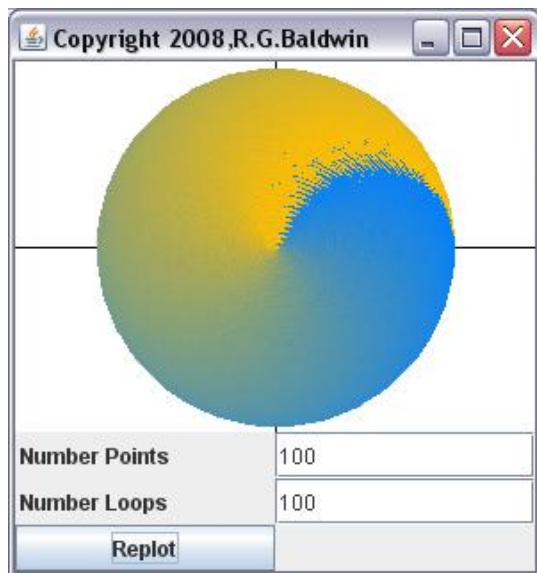
[Figure 7](#) shows the result of drawing lines in a more complex pattern among 25 points that are equally spaced around the circumference of the circle.

Figure 7 String art with 25 vertices and 11 loops.



[Figure 8](#) shows the result of drawing lines in an extremely complex pattern among 100 points that are equally spaced around the circumference of the circle.

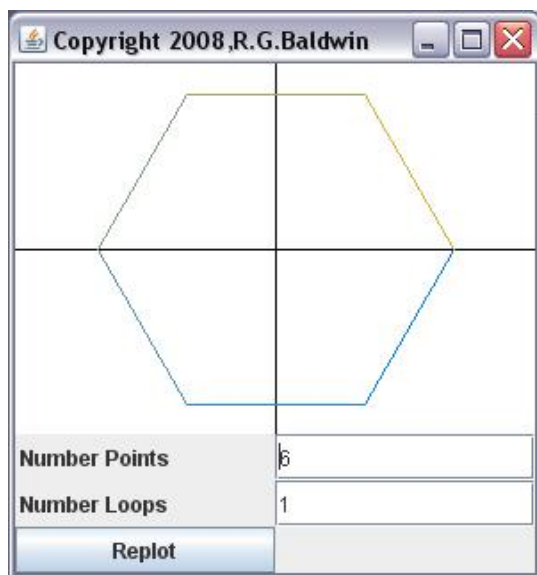
Figure 8 String art with 100 vertices and 100 loops.



As you can see, there were so many lines and they were so close together in [Figure 8](#) that the visual distinction between lines almost completely disappeared.

Finally, [Figure 9](#) shows the program output at startup, with six vertices and one loop.

Figure 9 Output from StringArt01 at startup.



Will discuss in fragments

As is my custom, I will discuss and explain this program in fragments. Much of the code in this program is very similar to code that I have explained earlier, so I won't repeat those explanations. Rather, I will concentrate on the code that is new and

different in this program. A complete listing of the program is provided in [Listing 21](#) near the end of the module.

Most of the code that is new to this program is in the method named **drawOffScreen**, which begins in [Listing 14](#).

Listing 14 . Beginning of the drawOffScreen method in StringArt01.

```
void drawOffScreen(Graphics2D g2D){
    //Erase the off-screen image and draw new axes,
but
    // don't move the origin.
    setCoordinateFrame(g2D, false);

    //Create a set of Point objects that specify
    // locations on the circumference of a circle and
    // save references to the Point objects in an
array.
    for(int cnt = 0; cnt < numberPoints; cnt++){
        points[cnt] = new GM2D04.Point(
            new GM2D04.ColMatrix(

100*Math.cos((cnt*360/numberPoints)

*Math.PI/180),

100*Math.sin((cnt*360/numberPoints)

*Math.PI/180)));
    } //end for loop
```

Even some of the code in the method named **drawOffScreen** was explained earlier, and that is the case for the code in [Listing 14](#).

Implement the algorithm that draws the lines

[Listing 15](#) begins the outer loop in a pair of nested loops that implement the algorithm to draw the lines shown in [Figure 6](#) through [Figure 9](#).

Listing 15 . Implement the algorithm that draws the lines.

```
GM2D04.Line line;

//Begin the outer loop.
for(int loop = 1;loop < loopLim;loop++){
    //The following variable specifies the array
    // element containing a point on which a line
will    // start.
        int start = -1;

        //The following variable specifies the number
of        // points that will be skipped between the
starting // point and the ending point for a line.
        int skip = loop;
        //The following logic causes the element number
to        // wrap around when it reaches the end of the
        // array.
        while(skip >= 2*numberPoints-1){
            skip -= numberPoints;
        }//end while loop

        //The following variable specifies the array
will    // element containing a point on which a line
        // end.
        int end = start + skip;
```

The algorithm

As I mentioned earlier, the algorithm goes something like the following:

- Draw a line connecting every point to its immediate neighbor on the circumference of the circle.
- Draw a line connecting every other point on the circumference of the circle.
- Draw a line connecting every third point on the circumference of the circle.
- Continue this process until the number of iterations satisfies the number of *Loops* specified by the user.

The code in the outer loop that begins in [Listing 15](#) is responsible for identifying the beginning and ending points for the lines that will be drawn during one iteration of the outer loop. Given the above information and the embedded comments in [Listing 15](#), you should have no difficulty understanding the logic in [Listing 15](#).

Draw the lines

The inner loop in the pair of nested loops is shown in [Listing 16](#). This loop constructs a series of **GM2D04.Line** objects and then causes visual manifestations of those objects to be drawn on the off-screen image.

Listing 16 . Draw the lines.

Listing 16 . Draw the lines.

```
for(int cnt = 0;cnt < numberPoints;cnt++){
    if(start < numberPoints-1){
        start++;
    }else{
        //Wrap around
        start -= (numberPoints-1);
    }//end else

    if(end < numberPoints-1){
        end++;
    }else{
        //Wrap around.
        end -= (numberPoints-1);
    }//end else

    //Create some interesting colors.
    g2D.setColor(new Color(cnt*255/numberPoints,
127+cnt*64/numberPoints,
                                255-
cnt*255/numberPoints));

    //Create a line and draw it.
    line = new
GM2D04.Line(points[start],points[end]);
    line.draw(g2D);
} //end inner loop
} //end outer loop
} //end drawOffScreen
```

Once again, given what you now know, and given the embedded comments in the code, you should have no difficulty understanding the logic of the code. Note in particular the requirement to wrap around when the element number exceeds the length of the array containing references to the **GM2D04.Point** objects that specify the locations of the vertices of the geometric object.

End of the discussion of StringArt01

That ends the discussion of the program named **StringArt01** .. It also ends the discussion of all five of the programs that I promised to explain in this module.

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter 5 - Vector Direction* .

The homework assignment for the next module is to study the Kjell tutorial through *Chapter 6 - Scaling and Unit Vectors* .

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 18](#) through [Listing 21](#) . Compile the code and execute it in conjunction with the game-math library provided in [Listing 17](#) . Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, I presented and explained four programs that use the game-math library named **GM2D04** . The purpose of the program named **VectorAdd05** is to use the **addVectorToPoint** method of the **GM2D04.Point** class to translate a geometric object from one location in space to a different location in space.

The purpose of the program named **VectorAdd05a** is to accomplish the same translation operation, but to do so in a possibly more efficient manner.

The purpose of the program named **VectorAdd06** is to teach you how to do rudimentary animation using the game-math library.

The purpose of the program named **StringArt01** is to teach you how to use methods of the game-math library to produce relatively complex drawings.

All of the programs are interactive in that they provide a GUI that allows the user to modify certain aspects of the behavior of the program.

What's next?

In the next module, you will learn how to update the game-math library to support 3D math, how to program the equations for projecting a 3D world onto a 2D plane, and how to add vectors in 3D.

You will also learn about scaling, translation, and rotation of a point in both 2D and 3D, about the rotation equations and how to implement them in both 2D and 3D, and much more.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0130: Putting the Game-Math Library to Work
- File: Game0130.htm
- Published: 10/16/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 17](#) through [Listing 21](#) below.

Listing 17 . Source code for the game-math library named GM2D04.

```
/*GM2D04.java  
Copyright 2008, R.G.Baldwin  
Revised 02/08/08
```

The name GM2Dnn is an abbreviation for GameMath2Dnn.

See the file named GM2D01.java for a general description of this game-math library file. This file is an update of GM2D03.

This update added the following new capabilities:

Vector addition - Adds this Vector object to a Vector object received as an incoming parameter and returns the sum as a resultant Vector object.

Added a method named getLength that returns the length of a vector as type double.

Added a method named addVectorToPoint to add a Vector to a Point producing a new Point.

Tested using JDK 1.6 under WinXP.

```
*****  
/  
import java.awt.geom.*;  
import java.awt.*;  
  
public class GM2D04{
```

```

    //An object of this class represents a 2D column
matrix.
    // An object of this class is the fundamental building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix{
        double[] data = new double[2];

        public ColMatrix(double data0,double data1){
            data[0] = data0;
            data[1] = data1;
        }//end constructor
        //-----
-//

        public String toString(){
            return data[0] + "," + data[1];
        }//end overridden toString method
        //-----
-//

        public double getData(int index){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                return data[index];
            }//end else
        }//end getData method
        //-----
-//

        public void setData(int index,double data){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                this.data[index] = data;
            }//end else
        }//end setData method
        //-----
-//

        //This method overrides the equals method inherited

```

```
// from the class named Object. It compares the
values
// stored in two matrices and returns true if the
// values are equal or almost equal and returns false
// otherwise.
public boolean equals(Object obj){
    if(obj instanceof GM2D04.ColMatrix &&
        Math.abs(((GM2D04.ColMatrix)obj).getData(0) -
                    getData(0)) <= 0.00001
&&
        Math.abs(((GM2D04.ColMatrix)obj).getData(1) -
                    getData(1)) <= 0.00001)
{
    return true;
}else{
    return false;
} //end else

} //end overridden equals method
//-----
-//

//Adds one ColMatrix object to another ColMatrix
// object, returning a ColMatrix object.
public GM2D04.ColMatrix add(GM2D04.ColMatrix matrix){
    return new GM2D04.ColMatrix(
        getData(0)+matrix.getData(0),
getData(1)+matrix.getData(1));
} //end subtract
//-----
-//

//Subtracts one ColMatrix object from another
// ColMatrix object, returning a ColMatrix object.
The
// object that is received as an incoming parameter
// is subtracted from the object on which the method
// is called.
public GM2D04.ColMatrix subtract(
        GM2D04.ColMatrix matrix)
{
```

```

        return new GM2D04.ColMatrix(
                                getData(0)-matrix.getData(0),
                                getData(1)-
matrix.getData(1));
        }//end subtract
        //-----
-//
    }//end class ColMatrix

//=====//

    public static class Point{
        GM2D04.ColMatrix point;

        public Point(GM2D04.ColMatrix point){//constructor
            //Create and save a clone of the ColMatrix object
            // used to define the point to prevent the point
            // from being corrupted by a later change in the
            // values stored in the original ColMatrix object
            // through use of its set method.
            this.point =
                new
ColMatrix(point.getData(0),point.getData(1));
        }//end constructor
        //-----
-//

        public String toString(){
            return point.getData(0) + "," + point.getData(1);
        }//end toString
        //-----
-//

        public double getData(int index){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                return point.getData(index);
            }//end else
        }//end getData
        //-----
-//

```

```

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            point.setData(index,data);
        }//end else
    }//end setData
    //-----
-//

    //This method draws a small circle around the
location
    // of the point on the specified graphics context.
    public void draw(Graphics2D g2D){
        Ellipse2D.Double circle =
            new
Ellipse2D.Double(getData(0)-3,
getData(1)-3,
6,
6);

        g2D.draw(circle);
    }//end draw
    //-----
-//

    //Returns a reference to the ColMatrix object that
    // defines this Point object.
    public GM2D04.ColMatrix getColMatrix(){
        return point;
    }//end getColMatrix
    //-----
-//

    //This method overrides the equals method inherited
    // from the class named Object. It compares the
values
    // stored in the ColMatrix objects that define two
    // Point objects and returns true if they are equal
    // and false otherwise.
    public boolean equals(Object obi){

```

```

        if(point.equals(((GM2D04.Point)obj).
                                getColMatrix()))
{
    return true;
}else{
    return false;
} //end else

} //end overridden equals method
//-----
-//

//Gets a displacement vector from one Point object to
// a second Point object. The vector points from the
// object on which the method is called to the object
// passed as a parameter to the method. Kjell
// describes this as the distance you would have to
// walk along the x and then the y axes to get from
// the first point to the second point.
public GM2D04.Vector getDisplacementVector(
                                GM2D04.Point point)
{
    return new GM2D04.Vector(new GM2D04.ColMatrix(
                                point.getData(0)-getData(0),
                                point.getData(1)-
getData(1)));
} //end getDisplacementVector
//-----
-//

//Adds a Vector to a Point producing a new Point.
public GM2D04.Point addVectorToPoint(
                                GM2D04.Vector vec){
    return new GM2D04.Point(new GM2D04.ColMatrix(
                                getData(0) + vec.getData(0),
                                getData(1) + vec.getData(1)));
} //end addVectorToPoint
//-----
-//
} //end class Point

//=====//

```

```

public static class Vector{
    GM2D04.ColMatrix vector;

    public Vector(GM2D04.ColMatrix vector){//constructor
        //Create and save a clone of the ColMatrix object
        // used to define the vector to prevent the vector
        // from being corrupted by a later change in the
        // values stored in the original ColVector object.
        this.vector = new ColMatrix(
vector.getData(0),vector.getData(1));
        }//end constructor
        //-----
-//

        public String toString(){
            return vector.getData(0) + "," + vector.getData(1);
        }//end toString
        //-----
-//

        public double getData(int index){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                return vector.getData(index);
            }//end else
        }//end getData
        //-----
-//

        public void setData(int index,double data){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                vector.setData(index,data);
            }//end else
        }//end setData
        //-----
-//

```

```

        //This method draws a vector on the specified
graphics
        // context, with the tail of the vector located at a
        // specified point, and with a small circle at the
        // head.
        public void draw(Graphics2D g2D, GM2D04.Point tail){
            Line2D.Double line = new Line2D.Double(
                tail.getData(0),
                tail.getData(1),
                tail.getData(0)+vector.getData(0),
tail.getData(1)+vector.getData(1));

            //Draw a small circle to identify the head.
            Ellipse2D.Double circle = new Ellipse2D.Double(
tail.getData(0)+vector.getData(0)-2,
tail.getData(1)+vector.getData(1)-2,
                4,
                4);
            g2D.draw(circle);
            g2D.draw(line);
        } //end draw
        //-----
-//

        //Returns a reference to the ColMatrix object that
        // defines this Vector object.
        public GM2D04.ColMatrix getColMatrix(){
            return vector;
        } //end getColMatrix
        //-----
-//

        //This method overrides the equals method inherited
        // from the class named Object. It compares the
values
        // stored in the ColMatrix objects that define two
        // Vector objects and returns true if they are equal
        // and false otherwise.
        public boolean equals(Object obi){

```



```

        if(vector.equals((
                                (GM2D04.Vector)obj).getColMatrix()))
    {
        return true;
    }else{
        return false;
    }//end else

    }//end overridden equals method
    //-----
-//

    //Adds this vector to a vector received as an
incoming
    // parameter and returns the sum as a vector.
    public GM2D04.Vector add(GM2D04.Vector vec){
        return new GM2D04.Vector(new ColMatrix(
                                vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1)));
    }//end add
    //-----
-//

    //Returns the length of a Vector object.
    public double getLength(){
        return Math.sqrt(
                                getData(0)*getData(0) +
getData(1)*getData(1));
    }//end getLength
    //-----
-//
    }//end class Vector

//=====//

```

```

//A line is defined by two points. One is called the
// tail and the other is called the head.

```

```

public static class Line{
    GM2D04.Point[] line = new GM2D04.Point[2];

```

```

    public Line(GM2D04.Point tail, GM2D04.Point head){
        //Create and save clones of the points used to
        // define the line to prevent the line from being
        // corrupted by a later change in the coordinate
        // values of the points.
        this.line[0] = new Point(new GM2D04.ColMatrix(
tail.getData(0),tail.getData(1)));
        this.line[1] = new Point(new GM2D04.ColMatrix(
head.getData(0),head.getData(1)));
    }//end constructor
    //-----
-//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + ","
            + line[1].getData(1);
    }//end toString
    //-----
-//

    public GM2D04.Point getTail(){
        return line[0];
    }//end getTail
    //-----
-//

    public GM2D04.Point getHead(){
        return line[1];
    }//end getHead
    //-----
-//

    public void setTail(GM2D04.Point newPoint){
        //Create and save a clone of the new point to
        // prevent the line from being corrupted by a
        // later change in the coordinate values of the
        // point.
        this.line[0] = new Point(new GM2D04.ColMatrix(

```

```

        newPoint.getData(0), newPoint.getData(1)));
    } //end setTail
    //-----
-//


    public void setHead(GM2D04.Point newPoint){
        //Create and save a clone of the new point to
        // prevent the line from being corrupted by a
        // later change in the coordinate values of the
        // point.
        this.line[1] = new Point(new GM2D04.ColMatrix(
            newPoint.getData(0), newPoint.getData(1)));
    } //end setHead
    //-----
-//

    public void draw(Graphics2D g2D){
        Line2D.Double line = new Line2D.Double(
            getTail().getData(0),
            getTail().getData(1),
            getHead().getData(0),
            getHead().getData(1));
        g2D.draw(line);
    } //end draw
    //-----
-//
    } //end class Line

//=====//

} //end class GM2D04
//=====//
/

```



Listing 18 . Source code for the sample program named VectorAdd05.

```

/*VectorAdd05.java
Copyright 2008, R.G.Baldwin
Revised 02/11/08

```

The purpose of this program is to use the `addVectorToPoint` method of the `GM2D04.Point` class to translate a geometric object from one location in space to a different location in space. Along the way, the program uses various methods of the classes in the game-math library named `GM2D04` to accomplish its purpose.

The program initially constructs and draws a black hexagon centered on the origin. The six points that define the vertices of the hexagon lie on a circle with a radius of 50 units. The points at the vertices and the lines that connect the points are drawn.

In addition, the program initially causes the hexagon to be translated by 50 units in the positive X direction and 50 units in the positive Y. The translated hexagon is drawn in red. The original black hexagon is not erased when the translated version is drawn in red.

A GUI is provided that allows the user to specify the following items and click a Replot button to cause the drawing to change:

Number points

X-component of the displacement vector.

Y-component of the displacement vector.

A checkbox to specify whether points are to be drawn.

A checkbox to specify whether lines are to be drawn.

Changing the number of points causes the number of vertices that describe the geometric object to change. For a large number of points, the geometric object becomes a circle. For only three points, it becomes a triangle. For four points, it becomes a square. For two points, it becomes a line, etc.

Changing the components of the displacement vector causes the geometric object to be translated to a different

location before being drawn in red.

Checking and unchecking the checkboxes causes the points and/or the lines to either be drawn or not drawn.

Tested using JDK 1.6 under WinXP.

```
/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;

class VectorAdd05{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class VectorAdd05
//=====/
/
```

```
class GUI extends JFrame implements ActionListener{
    //Specify the horizontal and vertical size of a JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//Off-screen graphics context.

    //The following two variables are used to establish the
    // location of the origin.
    double xAxisOffset;
    double yAxisOffset;

    int numberPoints = 6;//Can be modified by the user.
    JTextField numberPointsField; //User input field.
    //The components of the following displacement vector
    // can be modified by the user.
    GM2D04.Vector vector =
```

```

        new GM2D04.Vector(new
GM2D04.ColMatrix(50,50));
    JTextField vectorX;//User input field.
    JTextField vectorY;//User input field.

    //The following variables are used to determine whether
    // to draw the points and/or the lines.
    boolean drawPoints = true;
    boolean drawLines = true;
    Checkbox drawPointsBox;//User input field
    Checkbox drawLinesBox;//User input field.

    //The following variables are used to refer to array
    // objects containing the points that define the
    // vertices of the geometric object.
    GM2D04.Point[] points;
    GM2D04.Point[] newPoints;
    //-----
-//

GUI(){//constructor
    //Instantiate the array objects that will be used to
    // store the points that define the vertices of the
    // geometric object.
    points = new GM2D04.Point[numberPoints];
    newPoints = new GM2D04.Point[numberPoints];

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate the user input components.
    numberPointsField =new JTextField("6");
    vectorX = new JTextField("50");
    vectorY = new JTextField("50");
    drawPointsBox = new Checkbox("Draw Points",true);
    drawLinesBox = new Checkbox("Draw Lines",true);
    JButton button = new JButton("Replot");

    //Instantiate a JPanel that will house the user input
    // components and set its layout manager.

```

```

JPanel controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(3,3));

//Add the user input component and appropriate labels
// to the control panel.
controlPanel.add(new JLabel(" Number Points"));
controlPanel.add(numberPointsField);
controlPanel.add(drawPointsBox);
controlPanel.add(new JLabel(" Vector X"));
controlPanel.add(vectorX);
controlPanel.add(drawLinesBox);
controlPanel.add(new JLabel(" Vector Y"));
controlPanel.add(vectorY);
controlPanel.add(button);

//Add the control panel to the SOUTH position in the
// JFrame.
this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

//Create a new drawing canvas and add it to the
// CENTER of the JFrame above the control panel.
myCanvas = new MyCanvas();
this.getContentPane().add(
BorderLayout.CENTER,myCanvas);

//This object must be visible before you can get an
// off-screen image. It must also be visible before
// you can compute the size of the canvas.
setVisible(true);

//Make the size of the off-screen image match the
// size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

//Set the values that will be used to establish the
// origin, thus defining a coordinate frame.
xAxisOffset = osiWidth/2;
yAxisOffset = osiHeight/2;

```

```

//Create an off-screen image and get a graphics
// context on it.
osi = createImage(osiWidth,osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

//Erase the off-screen image and draw the axes
setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,true);

//Create the Point objects that define the geometric
// object and manipulate them to produce the desired
// results.
drawOffScreen(g2D);

//Register this object as an action listener on the
// button.
button.addActionListener(this);

//Cause the overridden paint method belonging to
// myCanvas to be executed.
myCanvas.repaint();

} //end constructor
//-----
-//

//The purpose of this method is to create the Point
// objects that define the vertices of the geometric
// object and manipulate them to produce the desired
// results.
void drawOffScreen(Graphics2D g2D){
    //Erase the off-screen image and draw new axes, but
    // don't change the coordinate frame.

setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,false);

    //Create a set of Point objects that specify
    // locations on the circumference of a circle and
    // save references to the Point objects in an array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM2D04.Point(
            new GM2D04.ColMatrix(

```



```

50*Math.cos((cnt*360/numberPoints)
                                     *Math.PI/180),

50*Math.sin((cnt*360/numberPoints)
*Math.PI/180)));
    if(drawPoints){//Draw points if true
        points[cnt].draw(g2D);
    }//end if
} //end for loop

GM2D04.Line line;
if(drawLines){//Instantiate and draw lines if true.
    for(int cnt = 0;cnt < numberPoints-1;cnt++){
        //Begin by drawing all of the lines but one.
        line = new
GM2D04.Line(points[cnt],points[cnt+1]);
        line.draw(g2D);
    }//end for loop
    //Draw the remaining line required to close the
    // polygon.
    line = new GM2D04.Line(points[numberPoints-1],
                           points[0]);
    line.draw(g2D);
} //end if

g2D.setColor(Color.RED);//Change drawing color to
RED.

//Translate the geometric object and save the points
// that define the translated object in another
array.
for(int cnt = 0;cnt < numberPoints;cnt++){
    newPoints[cnt] =

points[cnt].addVectorToPoint(vector);
    if(drawPoints){//Draw points if true.
        newPoints[cnt].draw(g2D);
    }//end if
} //end for loop

```

```

        if(drawLines){//Instantiate and draw lines if true.
            for(int cnt = 0;cnt < numberPoints-1;cnt++){
                line = new GM2D04.Line(newPoints[cnt],
newPoints[cnt+1]);
                line.draw(g2D);
            }//end for loop
            //Draw the remaining line required to close the
            // polygon.
            line = new GM2D04.Line(newPoints[numberPoints-1],
newPoints[0]);
            line.draw(g2D);
        }//end if

    }//end drawOffScreen
    //-----
-//

```

```

//This method is used to set the coordinate frame of
// the off-screen image by setting the origin to the
// specified offset values relative to origin of the
// world. The origin of the world is the upper-left
// corner of the off-screen image.
//The method draws black orthogonal axes on the
// off-screen image.
//There is no intention to perform mathematical
// operations on the axes, so they are drawn
// independently of the classes and methods in the
// game-math library using the simplest available
method
// for drawing lines.
//The method assumes that the origin is at the
// upper-left corner when the method is first called.
//Each time the method is called, it paints the
// background white erasing anything already there.
//The fourth parameter is used to determine if the
// origin should be translated by the values of the
// second and third parameters.
private void setCoordinateFrame(Graphics2D g2D,
                                double xOffset,
                                double yOffset,

```

```

boolean translate){
//Paint the background white
g2D.setColor(Color.WHITE);
g2D.fillRect(-(int)xOffset,-(int)yOffset,
              (int)osiWidth,(int)osiHeight);

//Translate the origin by the specified amount if the
// fourth parameter is true.
if(translate){
    g2D.translate((int)xOffset,(int)yOffset);
}//end if

//Draw new X and Y-axes in BLACK
g2D.setColor(Color.BLACK);
g2D.drawLine(-(int)(xOffset - 10),0,
              (int)(xOffset - 10),0);

g2D.drawLine(0,-(int)(yOffset - 10),
              0,(int)(yOffset - 10));

}//end setCoordinateFrame method
//-----
-//

//This method is called to respond to a click on the
// button.
public void actionPerformed(ActionEvent e){
    //Get user input values and use them to modify
several
    // values that control the translation and the
    // drawing.
    numberPoints = Integer.parseInt(
numberPointsField.getText());
    vector.setData(
0,Double.parseDouble(vectorX.getText()));
    vector.setData(
1,Double.parseDouble(vectorY.getText()));

    if(drawPointsBox.getState()){

```

```

        drawPoints = true;
    }else{
        drawPoints = false;
    }//end else

    if(drawLinesBox.getState()){
        drawLines = true;
    }else{
        drawLines = false;
    }//end else

    //Instantiate two new array objects with a length
    // that matches the new value for numberPoints.
    points = new GM2D04.Point[numberPoints];
    newPoints = new GM2D04.Point[numberPoints];

    //Draw a new off-screen image based on user inputs.
    drawOffScreen(g2D);
    myCanvas.repaint();//Copy off-screen image to canvas.
} //end actionPerformed

//=====//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method will be
    // called when the JFrame and the Canvas appear on
the
    // screen or when the repaint method is called on the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    } //end overridden paint()

} //end inner class MyCanvas

} //end class
GUI//=====
==//

```

Listing 19 . Source code for the program named VectorAdd05a.

```
/*VectorAdd05a.java  
Copyright 2008, R.G.Baldwin  
Revised 02/19/08
```

This program is a modified version of VectorAdd05 that uses a different scheme for modifying the vertices of the geometric object.

Note that the comments in this program were not updated to reflect these modifications. In particular, this version does not call the method named addVectorToPoint.

The purpose of this program is to use the addVectorToPoint method of the GM2D04.Point class to translate a geometric object from one location in space to a different location in space. Along the way, the program uses various methods of the classes in the game math library named GM2D04 to accomplish its purpose.

The program initially constructs and draws a black hexagon centered on the origin. The six points that define the vertices of the hexagon lie on a circle with a radius of 50 units. The points at the vertices and the lines that connect the points are drawn.

In addition, the program initially causes the hexagon to be translated by 50 units in the positive X direction and 50 units in the positive Y. The translated hexagon is drawn in red. The original black hexagon is not erased when the translated version is drawn in red.

A GUI is provided that allows the user to specify the following items and click a Replot button to cause the drawing to change:

Number points

X-component of the displacement vector.

Y-component of the displacement vector.

A checkbox to specify whether points are to be drawn.

A checkbox to specify whether lines are to be drawn.

Changing the number of points causes the number of vertices that describe the geometric object to change. For a large number of points, the geometric object becomes

a circle. For only three points, it becomes a triangle.

For four points, it becomes a square. For two points, it becomes a line, etc.

Changing the components of the displacement vector causes the geometric object to be translated to a different location before being drawn in red.

Checking and unchecking the checkboxes causes the points and/or the lines to either be drawn or not drawn.

Tested using JDK 1.6 under WinXP.

/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;
```

```
class VectorAdd05a{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    } //end main
```

```
} //end controlling class VectorAdd05a
```

```
//=====
```

/

```
class GUI extends JFrame implements ActionListener{
    //Specify the horizontal and vertical size of a JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
```

```

Image osi;//an off-screen image
int osiWidth;//off-screen image width
int osiHeight;//off-screen image height
MyCanvas myCanvas;//a subclass of Canvas
Graphics2D g2D;//Off-screen graphics context.

//The following two variables are used to establish the
// location of the origin.
double xAxisOffset;
double yAxisOffset;

int numberPoints = 6;//Can be modified by the user.
JTextField numberPointsField; //User input field.
    //The components of the following displacement vector
// can be modified by the user.
GM2D04.Vector vector =
    new GM2D04.Vector(new
GM2D04.ColMatrix(50,50));
JTextField vectorX;//User input field.
JTextField vectorY;//User input field.

//The following variables are used to determine whether
// to draw the points and/or the lines.
boolean drawPoints = true;
boolean drawLines = true;
Checkbox drawPointsBox;//User input field
Checkbox drawLinesBox;//User input field.

//The following variables are used to refer to array
// objects containing the points that define the
// vertices of the geometric object.
GM2D04.Point[] points;
//  GM2D04.Point[] newPoints;  //-----
-----//

GUI(){//constructor
    //Instantiate the array objects that will be used to
    // store the points that define the vertices of the
    // geometric object.
    points = new GM2D04.Point[numberPoints];
//    newPoints = new GM2D04.Point[numberPoints];

```

```

//Set JFrame size, title, and close operation.
setSize(hSize,vSize);
setTitle("Copyright 2008,R.G.Baldwin");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Instantiate the user input components.
numberPointsField =new JTextField("6");
vectorX = new JTextField("50");
vectorY = new JTextField("50");
drawPointsBox = new Checkbox("Draw Points",true);
drawLinesBox = new Checkbox("Draw Lines",true);
JButton button = new JButton("Replot");

//Instantiate a JPanel that will house the user input
// components and set its layout manager.
JPanel controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(3,3));

//Add the user input component and appropriate labels
// to the control panel.
controlPanel.add(new JLabel(" Number Points"));
controlPanel.add(numberPointsField);
controlPanel.add(drawPointsBox);
controlPanel.add(new JLabel(" Vector X"));
controlPanel.add(vectorX);
controlPanel.add(drawLinesBox);
controlPanel.add(new JLabel(" Vector Y"));
controlPanel.add(vectorY);
controlPanel.add(button);

//Add the control panel to the SOUTH position in the
// JFrame.
this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

//Create a new drawing canvas and add it to the
// CENTER of the JFrame above the control panel.
myCanvas = new MyCanvas();
this.getContentPane().add(
BorderLayout.CENTER,myCanvas);

```



```

//This object must be visible before you can get an
// off-screen image. It must also be visible before
// you can compute the size of the canvas.
setVisible(true);

//Make the size of the off-screen image match the
// size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

//Set the values that will be used to establish the
// origin, thus defining a coordinate frame.
xAxisOffset = osiWidth/2;
yAxisOffset = osiHeight/2;

//Create an off-screen image and get a graphics
// context on it.
osi = createImage(osiWidth,osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

//Erase the off-screen image and draw the axes
setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,true);

//Create the Point objects that define the geometric
// object and manipulate them to produce the desired
// results.
drawOffScreen(g2D);

//Register this object as an action listener on the
// button.
button.addActionListener(this);

//Cause the overridden paint method belonging to
// myCanvas to be executed.
myCanvas.repaint();

} //end constructor
//-----
-//

```

//The purpose of this method is to Create the Point

```

// objects that define the vertices of the geometric
// object and manipulate them to produce the desired
// results.
void drawOffScreen(Graphics2D g2D){
    //Erase the off-screen image and draw new axes, but
    // don't change the coordinate frame.

setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,false);

    //Create a set of Point objects that specify
    // locations on the circumference of a circle and
    // save references to the Point objects in an array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM2D04.Point(
            new GM2D04.ColMatrix(

50*Math.cos((cnt*360/numberPoints)
                                                    *Math.PI/180),

50*Math.sin((cnt*360/numberPoints)

*Math.PI/180)));
        if(drawPoints){//Draw points if true
            points[cnt].draw(g2D);
        }//end if
    }//end for loop

    GM2D04.Line line;
    if(drawLines){//Instantiate and draw lines if true.
        for(int cnt = 0;cnt < numberPoints-1;cnt++){
            //Begin by drawing all of the lines but one.
            line = new
GM2D04.Line(points[cnt],points[cnt+1]);
            line.draw(g2D);
        }//end for loop
        //Draw the remaining line required to close the
        // polygon.
        line = new GM2D04.Line(points[numberPoints-1],
                                points[0]);

        line.draw(g2D);
    }//end if

```

```

    g2D.setColor(Color.RED); //Change drawing color to
    RED.

    //Translate the geometric object and save the
    points
    // that define the translated object in the same
    // array object.
    for(int cnt = 0; cnt < numberPoints; cnt++){
        points[cnt].setData(
            0, points[cnt].getData(0) +
vector.getData(0));
        points[cnt].setData(
            1, points[cnt].getData(1) +
vector.getData(1));

        if(drawPoints){ //Draw points if true.
            points[cnt].draw(g2D);
        } //end if
    } //end for loop

    if(drawLines){ //Instantiate and draw lines if true.
        for(int cnt = 0; cnt < numberPoints-1; cnt++){
            line = new GM2D04.Line(points[cnt],
                                   points[cnt+1]);

            line.draw(g2D);
        } //end for loop
        //Draw the remaining line required to close the
        // polygon.
        line = new GM2D04.Line(points[numberPoints-1],
                                points[0]);

        line.draw(g2D);
    } //end if

} //end drawOffScreen
//-----
-//

//This method is used to set the coordinate frame of
// the off-screen image by setting the origin to the
// specified offset values relative to origin of the
// world. The origin of the world is the upper-left
// corner of the off-screen image.

```

```

//The method draws black orthogonal axes on the
// off-screen image.
//There is no intention to perform mathematical
// operations on the axes, so they are drawn
// independently of the classes and methods in the
// game-math library using the simplest available
method
// for drawing lines.
//The method assumes that the origin is at the
// upper-left corner when the method is first called.
//Each time the method is called, it paints the
// background white erasing anything already there.
//The fourth parameter is used to determine if the
// origin should be translated by the values of the
// second and third parameters.
private void setCoordinateFrame(Graphics2D g2D,
                                double xOffset,
                                double yOffset,
                                boolean translate){

    //Paint the background white
    g2D.setColor(Color.WHITE);
    g2D.fillRect(-(int)xOffset, -(int)yOffset,
                 (int)osiWidth, (int)osiHeight);

    //Translate the origin by the specified amount if the
    // fourth parameter is true.
    if(translate){
        g2D.translate((int)xOffset, (int)yOffset);
    }//end if

    //Draw new X and Y-axes in BLACK
    g2D.setColor(Color.BLACK);
    g2D.drawLine(-(int)(xOffset - 10), 0,
                 (int)(xOffset - 10), 0);

    g2D.drawLine(0, -(int)(yOffset - 10),
                 0, (int)(yOffset - 10));

} //end setCoordinateFrame method
//-----
-//

```

```

    //This method is called to respond to a click on the
    // button.
    public void actionPerformed(ActionEvent e){
        //Get user input values and use them to modify
        several
        // values that control the translation and the
        // drawing.
        numberPoints = Integer.parseInt(

numberPointsField.getText());
        vector.setData(

0, Double.parseDouble(vectorX.getText()));
        vector.setData(

1, Double.parseDouble(vectorY.getText()));

        if(drawPointsBox.getState()){
            drawPoints = true;
        }else{
            drawPoints = false;
        }//end else

        if(drawLinesBox.getState()){
            drawLines = true;
        }else{
            drawLines = false;
        }//end else

        //Instantiate two new array objects with a length
        // that matches the new value for numberPoints.
        points = new GM2D04.Point[numberPoints];
        //
        newPoints = new GM2D04.Point[numberPoints];
        //Draw a new off-screen image based on user inputs.
        drawOffScreen(g2D);
        myCanvas.repaint();//Copy off-screen image to canvas.
    }//end actionPerformed

//=====//

```

```

//This is an inner class of the GUI class.

```

```

class MyCanvas extends Canvas{
    //Override the paint() method. This method will be
    // called when the JFrame and the Canvas appear on
the
    // screen or when the repaint method is called on the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    }//end overridden paint()

} //end inner class MyCanvas

} //end class GUI
//=====
/

```

Listing 20 . Source code for the program named VectorAdd06.

```

/*VectorAdd06.java
Copyright 2008, R.G.Baldwin
Revised 02/22/08.

```

This is an update to the program named VectorAdd05. The behavior of this program is similar to the earlier program except that instead of displaying a static view of

the translated geometric object when the Replot button is clicked, this program animates the geometric object causing it to move from its original location to its new location in 100 incremental steps along a straight line path.

The animation loop sleeps for ten milliseconds during each

of the 100 iterations. Therefore, approximately one second

(1000 milliseconds and possibly more) is required for the object to make the trip from its initial location to its new location. Once it reaches the new location, the program is ready for the user to change input values and

click the Replot button again.

As with the program named VectorAdd05, this program uses the addVectorToPoint method of the GM2D04.Point class to translate a geometric object from one location in space to a different location in space. In this program, however, the translation is performed in 100 incremental steps to produce an animation. The animated geometric object is drawn in red to make it visually distinct from the original object. The original object is not erased from the display.

The program initially constructs and draws a black hexagon in the upper-left corner of the canvas. The six points that define the vertices of the hexagon lie on a circle with a radius of 50 units. The points at the vertices and the lines that connect the points are drawn initially.

A GUI is provided that allows the user to specify the following items and click a Replot button to cause the animation to begin:

Number points

X-component of the displacement vector.

Y-component of the displacement vector.

A checkbox to specify whether points are to be drawn.

A checkbox to specify whether lines are to be drawn.

Changing the number of points causes the number of vertices that describe the geometric object to change. For a large number of points, the geometric object becomes a circle. For only three points, it becomes a triangle. For four points, it becomes a rectangle. For two points, it becomes a line, etc. On the computer that I am now using, the animation becomes jerky at about 700 points when both the points and the lines are drawn.

Changing the components of the displacement vector causes the geometric object to be translated to a different

location.

Checking and unchecking the checkboxes causes the points and/or the lines to either be drawn or not drawn.

Tested using JDK 1.6 under WinXP.

```
*****  
/  

```

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.geom.*;  
import java.awt.event.*;
```

```
class VectorAdd06{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class VectorAdd06  
//=====/  
/  

```

```
class GUI extends JFrame implements ActionListener{  
  
    //Specify the horizontal and vertical size of a JFrame  
    // object.  
    int hSize = 400;  
    int vSize = 400;  
    Image osi;//an off-screen image  
    int osiWidth;//off-screen image width  
    int osiHeight;//off-screen image height  
    MyCanvas myCanvas;//a subclass of Canvas  
    Graphics2D g2D;//off-screen graphics context.  
  
    //The following two variables are used to establish the  
    // location of the origin.  
    double xAxisOffset;  
    double yAxisOffset;  
  
    int numberPoints = 6;//Can be modified by the user.  
    JTextField numberPointsField; //User input field.  
    //The components of the following displacement vector
```



```

// can be modified by the user.
GM2D04.Vector vector =
    new GM2D04.Vector(new GM2D04.ColMatrix(0,0));
JTextField vectorX;//User input field.
JTextField vectorY;//User input field.

//The following variables are used to determine whether
// to draw the points and/or the lines.
boolean drawPoints = true;
boolean drawLines = true;
Checkbox drawPointsBox;//User input field
Checkbox drawLinesBox;//User input field.

//The following variables are used to refer to array
// objects containing the points that define the
// vertices of the geometric objects.
GM2D04.Point[] points;
GM2D04.Point[] newPoints;

boolean animation = false;
//-----
-//

GUI(){//constructor
    //Instantiate the array objects that will be used to
    // store the points that define the vertices of the
    // geometric object.
    points = new GM2D04.Point[numberPoints];
    newPoints = new GM2D04.Point[numberPoints];

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate the user input components.
    numberPointsField =new JTextField("6");
    vectorX = new JTextField("300");
    vectorY = new JTextField("100");
    drawPointsBox = new Checkbox("Draw Points",true);
    drawLinesBox = new Checkbox("Draw Lines",true);
    JButton button = new JButton("Replot");

```

```

//Instantiate a JPanel that will house the user input
// components and set its layout manager.
JPanel controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(3,3));

//Add the user input component and appropriate labels
// to the control panel.
controlPanel.add(new JLabel(" Number Points"));
controlPanel.add(numberPointsField);
controlPanel.add(drawPointsBox);
controlPanel.add(new JLabel(" Vector X"));
controlPanel.add(vectorX);
controlPanel.add(drawLinesBox);
controlPanel.add(new JLabel(" Vector Y"));
controlPanel.add(vectorY);
controlPanel.add(button);

//Add the control panel to the SOUTH position in the
// JFrame.
this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

//Instantiate a new drawing canvas and add it to the
// CENTER of the JFrame above the control panel.
myCanvas = new MyCanvas();
this.getContentPane().add(
BorderLayout.CENTER,myCanvas);

//This object must be visible before you can get an
// off-screen image. It must also be visible before
// you can compute the size of the canvas.
setVisible(true);

//Make the size of the off-screen image match the
// size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

//Set the values that will be used to establish the

```

```

// origin, thus defining a coordinate frame.
xAxisOffset = osiWidth/2;
yAxisOffset = osiHeight/2;

//Create an off-screen image and get a graphics
// context on it.
osi = createImage(osiWidth,osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

//Erase the off-screen image, establish the
// coordinate frame by translating the origin to the
// new location and draw the axes
setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,true);

//Create the Point objects that define the geometric
// object and manipulate them to produce the desired
// results.
drawOffScreen(g2D);

//Register this object as an action listener on the
// button.
button.addActionListener(this);

//Cause the overridden paint method belonging to
// myCanvas to be executed.
myCanvas.repaint();

} //end constructor
//-----
-//

//The purpose of this method is to Create the Point
// objects that define the vertices of the geometric
// object and manipulate them to produce the desired
// results.
void drawOffScreen(Graphics2D g2D){
    //Erase the off-screen image and draw new axes, but
    // don't change the coordinate frame.

setCoordinateFrame(g2D,xAxisOffset,yAxisOffset,false);

    //Create a set of Point objects that specify

```

```

    // locations on the circumference of a circle and
    // save references to the Point objects in an array.
    // Position the circle at the upper-left corner of
the
    // canvas.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM2D04.Point(
            new GM2D04.ColMatrix(

50*Math.cos((cnt*360/numberPoints)
                                *Math.PI/180) -
150,

50*Math.sin((cnt*360/numberPoints)
                                *Math.PI/180) -
100));
        if(drawPoints){//Draw points if true
            points[cnt].draw(g2D);
        }//end if
    }//end for loop

    GM2D04.Line line;
    if(drawLines){//Instantiate and draw lines if true.
        for(int cnt = 0;cnt < numberPoints-1;cnt++){
            //Begin by drawing all of the lines but one.
            line = new
GM2D04.Line(points[cnt],points[cnt+1]);
            line.draw(g2D);
        }//end for loop
        //Draw the remaining line required to close the
        // polygon.
        line = new GM2D04.Line(points[numberPoints-1],
                                points[0]);

        line.draw(g2D);
    }//end if

    g2D.setColor(Color.RED);//Change drawing color to
RED.

    if(animation){
        //This code is executed only after the Replot

```

```

button
    // has been clicked once.
    //Translate the geometric object and save the
points
    // that define the translated object in another
    // array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        newPoints[cnt] =

points[cnt].addVectorToPoint(vector);
        if(drawPoints){//Draw points if true.
            newPoints[cnt].draw(g2D);
        }//end if
    }//end for loop

    if(drawLines){//Instantiate and draw lines if true.
        for(int cnt = 0;cnt < numberPoints-1;cnt++){
            line = new GM2D04.Line(newPoints[cnt],

newPoints[cnt+1]);
            line.draw(g2D);
        }//end for loop
        //Draw the remaining line required to close the
        // polygon.
        line = new GM2D04.Line(newPoints[numberPoints-1],

newPoints[0]);
        line.draw(g2D);
    }//end if
} //end if(animation)

} //end drawOffScreen
//-----
-//

//This method is used to set the coordinate frame of
// the off-screen image by setting the origin to the
// specified offset values relative to the origin of
the
// world. The origin of the world is the upper-left
// corner of the off-screen image.
//The method draws black orthogonal axes on the

```

```

// off-screen image.
//There is no intention to perform mathematical
// operations on the axes, so they are drawn
// independently of the classes and methods in the
// game-math library using the simplest available
method
// for drawing lines.
//The method assumes that the origin is at the
// upper-left corner when the method is first called.
//Each time the method is called, it paints the
// background white erasing anything already there.
//The fourth parameter is used to determine if the
// origin should be translated by the values of the
// second and third parameters.
private void setCoordinateFrame(Graphics2D g2D,
                                double xOffset,
                                double yOffset,
                                boolean translate){

    //Paint the background white
    g2D.setColor(Color.WHITE);
    g2D.fillRect(-(int)xOffset, -(int)yOffset,
                 (int)osiWidth, (int)osiHeight);

    //Translate the origin by the specified amount if the
    // fourth parameter is true.
    if(translate){
        g2D.translate((int)xOffset, (int)yOffset);
    }//end if

    //Draw new X and Y-axes in BLACK
    g2D.setColor(Color.BLACK);
    g2D.drawLine(-(int)(xOffset - 10), 0,
                 (int)(xOffset - 10), 0);

    g2D.drawLine(0, -(int)(yOffset - 10),
                 0, (int)(yOffset - 10));

} //end setCoordinateFrame method
//-----
-//

```

//This method is called to respond to a click on the

```

// button.
public void actionPerformed(ActionEvent e){
    //Get user input values and use them to modify
several
    // values that control the translation and the
    // drawing.
    numberPoints = Integer.parseInt(
numberPointsField.getText());

    vector.setData(
0, Double.parseDouble(vectorX.getText()));
    vector.setData(
1, Double.parseDouble(vectorY.getText()));

    if(drawPointsBox.getState()){
        drawPoints = true;
    }else{
        drawPoints = false;
    }//end else

    if(drawLinesBox.getState()){
        drawLines = true;
    }else{
        drawLines = false;
    }//end else

    //Instantiate two new array objects with a length
    // that matches the new value for numberPoints.
    points = new GM2D04.Point[numberPoints];
    newPoints = new GM2D04.Point[numberPoints];

    //Set the animation flag to true to enable animation.
    animation = true;

    //Spawn an animation thread
    Animate animate = new Animate();
    animate.start();

} //end actionPerformed

```

```
//=====//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the update method to eliminate the default
    // clearing of the Canvas in order to reduce or
    // eliminate the flashing that that is often caused
by
    // such default clearing.
    //In this case, it isn't necessary to clear the
canvas
    // because the off-screen image is cleared each time
    // it is updated. This method will be called when the
    // JFrame and the Canvas appear on the screen or when
    // the repaint method is called on the Canvas object.
    public void update(Graphics g){
        paint(g);//Call the overridden paint method.
    }//end overridden update()

    //Override the paint() method. The purpose of the
    // paint method in this program is simply to copy the
    // off-screen image onto the canvas. This method is
    // called by the update method above.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    }//end overridden paint()

} //end inner class MyCanvas

//=====//

//This is an animation thread.
class Animate extends Thread{
    public void run(){
        //During each iteration of the following loop, an
        // incremental displacement vector is created with
        // X and Y components that are equal to 1/100 of
the
        // user-specified displacement vector multiplied by
        // the loop counter value.
        //The incremental displacement vector is used by
```


the

```
// code in the drawOffScreen method to translate
// the geometric object to a new location on the
// off-screen image. Then the repaint method is
// called on the canvas to cause the off-screen
// image to be copied onto the canvas. After that,
// this thread sleeps for ten milliseconds before
// starting the next iteration.
```

```
//The bottom line is that this code causes the
// geometric object to move in incremental steps
// from its initial location to a new location
```

based

```
// on a displacement vector specified by the user.
// In other words, the geometric object reaches its
// final location in a straight-line animated
// manner, taking 100 steps to get there.
```

```
//Compute the incremental distances that the
// geometric object will move during each
```

iteration.

```
double xInc = vector.getData(0)/100;
double yInc = vector.getData(1)/100;
```

```
//Do the animated move.
```

```
for(int cnt = 0;cnt < 100;cnt++){
    vector.setData(0,cnt*xInc);
    vector.setData(1,cnt*yInc);
```

```
//Draw a new off-screen image based on the
// incremental displacement vector and other user
// inputs.
```

```
drawOffScreen(g2D);
//Copy off-screen image to canvas.
myCanvas.repaint();
```

```
//Sleep for ten milliseconds.
```

```
try{
    Thread.currentThread().sleep(10);
}catch(InterruptedException ex){
    ex.printStackTrace();
```

```
}//end catch
```

```
}//end for loop
```


```

        }//end run
    }//end inner class named Animate

//=====//

}//end class
GUI//=====
==//

```



Listing 21 . Source code for the program named StringArt01.

```

/*StringArt01.java
Copyright 2008, R.G.Baldwin
Revised 02/20/08

```

This program produces a 2D string art image by connecting various points that are equally spaced on the circumference of a circle.

At startup, the program constructs and draws a multi-colored hexagon centered on the origin. The six points that define the vertices of the hexagon lie on a circle with a radius of 100 units. The points at the vertices are not drawn, but the lines that connect the vertices are drawn.

A GUI is provided that allows the user to specify the following items and click a Replot button to cause the drawing to change:

Number Points
Number Loops

Changing the number of points causes the number of vertices that describe the geometric object to change.

Changing the number of loops causes the number of lines that are drawn to connect the vertices to change. For a value of 1, each vertex is connected to the one next to it. For a value of 2, additional lines are drawn connecting every other vertex. For a value of 3,

connecting every second vertex. For a value of 3, additional lines are drawn connecting every third vertex, etc.

Making the number of points large and making the number of loops large produces many interesting patterns.

Tested using JDK 1.6 under WinXP.

```
*****
/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;

class StringArt01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class StringArt01
//=====/
/

class GUI extends JFrame implements ActionListener{
    //Specify the horizontal and vertical size of a JFrame
    // object.
    int hSize = 300;
    int vSize = 320;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//Off-screen graphics context.

    int numberPoints = 6;//Can be modified by the user.
    JTextField numberPointsField; //User input field.
    JTextField loopsField;//User input field.
    int loopLim = 2;

    //The following variable is used to refer to the array
    // object containing the points that define the
    // vertices of the geometric object.
    GM2D04.Point[] points;
```

```

        GM2D04.Point[] points;
        //-----
-//

GUI(){//constructor
    //Instantiate the array object that will be used to
    // store the points that define the vertices of the
    // geometric object.
    points = new GM2D04.Point[numberPoints];

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate the user input components.
    numberPointsField =new JTextField("6");
    loopsField = new JTextField("1");
    JButton button = new JButton("Replot");

    //Instantiate a JPanel that will house the user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(3,3));

    //Add the user input components and appropriate
labels
    // to the control panel.
    controlPanel.add(new JLabel(" Number Points"));
    controlPanel.add(numberPointsField);
    controlPanel.add(new JLabel(" Number Loops"));
    controlPanel.add(loopsField);
    controlPanel.add(button);

    //Add the control panel to the SOUTH position in the
    // JFrame.
    this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

    //Create a new drawing canvas and add it to the
    // CENTER of the JFrame above the control panel.
    mvCanvas = new MvCanvas();

```

```

        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER,myCanvas);

        //This object must be visible before you can get an
        // off-screen image. It must also be visible before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Erase the off-screen image and draw the axes
        setCoordinateFrame(g2D,true);

        //Create the Point objects that define the geometric
        // object and manipulate them to produce the desired
        // results.
        drawOffScreen(g2D);

        //Register this object as an action listener on the
        // button.
        button.addActionListener(this);

        //Cause the overridden paint method belonging to
        // myCanvas to be executed.
        myCanvas.repaint();

    } //end constructor
    //-----
-//

    //The purpose of this method is to Create the Point
    // objects that define the vertices of the geometric
    // object and manipulate them to produce the desired

```

```

// object and manipulate them to produce the desired
// results.
void drawOffScreen(Graphics2D g2D){
    //Erase the off-screen image and draw new axes, but
    // don't move the origin.
    setCoordinateFrame(g2D,false);

    //Create a set of Point objects that specify
    // locations on the circumference of a circle and
    // save references to the Point objects in an array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM2D04.Point(
            new GM2D04.ColMatrix(

100*Math.cos((cnt*360/numberPoints)

*Math.PI/180),

100*Math.sin((cnt*360/numberPoints)

*Math.PI/180)));
    }//end for loop

    //Implement the algorithm that draws the lines.
    GM2D04.Line line;

    //Begin the outer loop.
    for(int loop = 1;loop < loopLim;loop++){
        //The following variable specifies the array
        // element containing a point on which a line will
        // start.
        int start = -1;

        //The following variable specifies the number of
        // points that will be skipped between the starting
        // point and the ending point for a line.
        int skip = loop;
        //The following logic causes the element number to
        // wrap around when it reaches the end of the
        // array.
        while(skip >= 2*numberPoints-1){
            skip -= numberPoints;
        }//end while loop
    }
}

```

```

//end while loop

//The following variable specifies the array
// element containing a point on which a line will
// end.
int end = start + skip;

//Begin inner loop. This loop actually constructs
// the GM2D04.Line objects and causes visual
// manifestations of those objects to be drawn on
// the off-screen image. Note the requirement to
// wrap around when the element numbers exceed the
// length of the array.
for(int cnt = 0;cnt < numberPoints;cnt++){
    if(start < numberPoints-1){
        start++;
    }else{
        //Wrap around
        start -= (numberPoints-1);
    }//end else

    if(end < numberPoints-1){
        end++;
    }else{
        //Wrap around.
        end -= (numberPoints-1);
    }//end else

    //Create some interesting colors.
    g2D.setColor(new Color(cnt*255/numberPoints,
                            127+cnt*64/numberPoints,
                            255-
cnt*255/numberPoints));

    //Create a line and draw it.
    line = new
GM2D04.Line(points[start],points[end]);
    line.draw(g2D);
} //end inner loop
} //end outer loop
} //end drawOffScreen
//-----
-//

```

```

..

//This method is used to set the coordinate frame of
// the off-screen image by setting the origin to the
// center of the off-screen image and drawing black
// orthogonal axes that intersect at the origin.
//The second parameter is used to determine if the
// origin should be translated to the center.
private void setCoordinateFrame(Graphics2D g2D,
                                boolean translate){
    //Translate the origin to the center of the off-
screen
    // image if the fourth parameter is true.
    if(translate){
        g2D.translate((int)osiWidth/2,(int)osiHeight/2);
    }//end if

    //Paint the background white
    g2D.setColor(Color.WHITE);
    g2D.fillRect(-(int)osiWidth/2,-(int)osiHeight/2,
                (int)osiWidth,(int)osiHeight);

    //Draw new X and Y-axes in BLACK
    g2D.setColor(Color.BLACK);
    g2D.drawLine(-(int)(osiWidth/2),0,
                (int)(osiWidth/2),0);

    g2D.drawLine(0,-(int)(osiHeight/2),
                0,(int)(osiHeight/2));

} //end setCoordinateFrame method
//-----
-//

//This method is called to respond to a click on the
// button.
public void actionPerformed(ActionEvent e){
    //Get user input values and use them to modify
several
    // values that control the drawing.
    numberPoints = Integer.parseInt(
numberPointsField.getText());

```



```

numberPoints = Integer.parseInt(loopsField.getText());

    loopLim = Integer.parseInt(loopsField.getText()) + 1;

    //Instantiate a new array object with a length
    // that matches the new value for numberPoints.
    points = new GM2D04.Point[numberPoints];

    //Draw a new off-screen image based on user inputs.
    drawOffScreen(g2D);
    myCanvas.repaint();//Copy off-screen image to canvas.
} //end actionPerformed

//=====//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method will be
    // called when the JFrame and the Canvas appear on
the
    // screen or when the repaint method is called on the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    } //end overridden paint()

} //end inner class MyCanvas

} //end class GUI
//=====//
/

```

Exercises

Exercise 1

Using Java and the game-math library named **GM2D04** , , or using a different programming environment of your choice, write a program that begins with an image showing:

- Cartesian coordinate axes with the origin at the center of the image,
- a small black pentagon centered on the origin, and
- a button labeled **Replot**

No user input controls other than the button should be visible in your image.

Make each side of the pentagon approximately one-half inch in length as shown in [Figure 10](#).

Each time you click the **Replot** button, a red pentagon appears on top of the black pentagon and then moves smoothly to a random location somewhere in the image as shown in [Figure 11](#).

If you click the button more than once, the old red pentagon is erased and a new red pentagon appears and moves as described above.

Move the red pentagon in a straight line in approximately 100 incremental steps over a time period of approximately one second.

Cause the program to display your name in some manner.

Figure 10 Screen output from Exercise 1 at startup.

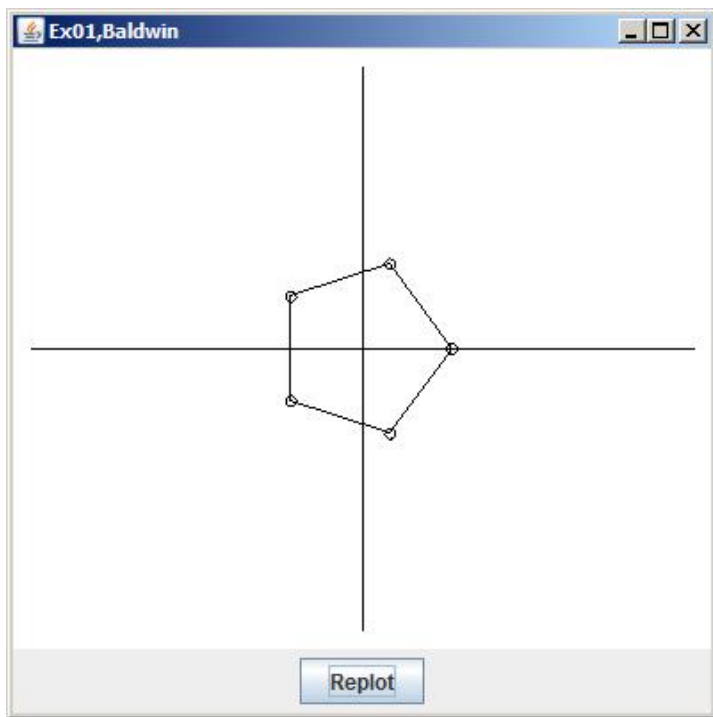
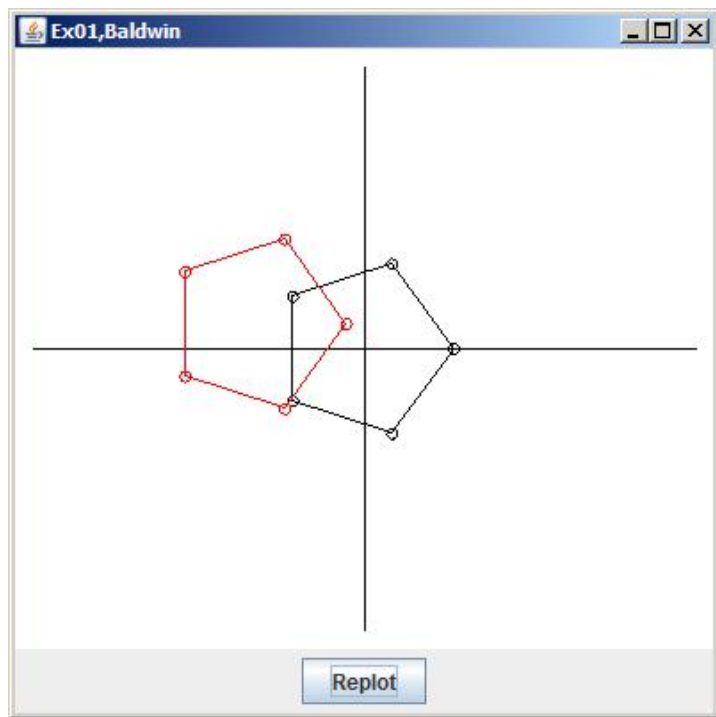


Figure 11 Screen output from Exercise 1 after clicking Replot button.



-end-

Game0130r: Review

This module contains review questions and answers keyed to the module titled [GAME 2302-0130: Putting the Game-Math Library to Work](#).

Table of Contents

- [Preface](#)
- [Questions](#)
 - [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#)
- [Answers](#)
- [Miscellaneous](#)

Preface

This module contains review questions and answers keyed to the module titled [GAME 2302-0130: Putting the Game-Math Library to Work](#).

The questions and the answers are connected by hyperlinks to make it easy for you to navigate from the question to the answer and back again.

Questions

Question 1 .

What is the length of the vector represented by $(3,0)^T$ where the upper-case T indicates "transpose" and is often shown as an upper-case superscript T.

- A. 2
- B. 3
- C. 4
- D. 5

[Answer 1](#)

Question 2

True or False: When a 2D vector is aligned with the x axis, the column matrix that represents it has a non-zero value in the first element, and zero for the second element.

[Answer 2](#)

Question 3

True or False: When a 2D vector is aligned with the y axis, the column matrix that represents it has a non-zero value in the first element, and zero for the second element.

[Answer 3](#)

Question 4

What is the length of the vector represented by $(0,4)^T$

- A. 2
- B. 3
- C. 4
- D. 5

[Answer 4](#)

Question 5

What is the length of the hypotenuse of a right triangle whose two sides are 6 and 8?

- A. 6

- B. 8
- C. 10
- D. 12

[Answer 5](#)

Question 6

What is the length of the vector represented by $(4, 3)^T$

- A. 3
- B. 4
- C. 5
- D. 6

[Answer 6](#)

Question 7

Given: The length of the vector v is often represented as $|v|$

Question: What is the value of $|(4, 3)^T|$

- A. 3
- B. 4
- C. 5
- D. 6

[Answer 7](#)

Question 8

True or False: Given vectors u and v , the following is a true statement where \geq indicates "greater than or equal"

$$|u + v| \geq |u| + |v|$$

[Answer 8](#)

Question 9

True or False: Three dimensional vectors have length.

[Answer 9](#)

Question 10

Given:

- The \wedge character represents exponentiation
- $\text{sqrt}()$ represents computation of the square root of the value contained in the parentheses.
- $(x,y,z)^T$ is a transpose column matrix that represents a three-dimensional vector

True or False: The following statement is true.

$$|(x,y,z)^T| = \text{sqrt}(x^3 + y^3 + z^3)$$

[Answer 10](#)

Question 11

What is the value of $|(2,-4,4)^T|$

- A. 4

- B. 5
- C. 6
- D. 7

[Answer 11](#)

Question 12

True or False: Length is a property of a vector and is not a property of the column matrix that is used to represent it.

[Answer 12](#)

Question 13

True or False: The length of a vector can be positive or negative.

[Answer 13](#)

Question 14

True or False: Two vectors that are equal to each other must have the same length.

[Answer 14](#)

Question 15

True or False: Two vectors that have the same length must be equal to each other.

[Answer 15](#)

Question 16

True or False: If v is a vector, then $-v$ is a vector that is perpendicular to v .

[Answer 16](#)

Question 17

Given: the vector represented by $(4,0)T$

True or False: The vector is parallel to the X axis of the coordinate frame we are using. Often this is horizontal, pointing right. (Or, you might say that the vector is oriented at 0 degrees if you consider due East to be 0 degrees.)

[Answer 17](#)

Question 18

Given: The orientation of a vector is usually expressed as an angle with the positive x axis of a coordinate frame. There are two ways of doing this:

- The angle is 0 degrees to 360 degrees measured as a counter-clockwise rotation from the positive x axis.
- The angle is 0 degrees to +180 degrees measured as a counter-clockwise rotation from the positive x axis, or is 0 degrees to -180 degrees measured as a clockwise rotation from the positive x axis.

Kjell, Chapter 5

What is the direction of the vector represented by $(5,-5)T$

- A. 0 degrees
- B. 45 degrees
- C. 315 degrees
- D. -45 degrees

[Answer 18](#)

Question 19

True or False: The formula for the direction of a 2D vector is:

angle of $(x, y)^T = \arctan(y/x)$

[Answer 19](#)

Question 20

What is the direction of the vector represented by $(5, -5)^T$

- A. 45 degrees
- B. 135 degrees
- C. -45 degrees
- D. -135 degrees

[Answer 20](#)

Question 21

What is the direction of the vector represented by $(-5, -5)^T$

- A. 45 degrees
- B. 135 degrees
- C. -45 degrees
- D. -135 degrees

[Answer 21](#)

Question 22

A vector has a length of 4 and an orientation of 150 degrees. Which of the following transpose matrices properly describes the vector?

- A. $(-3.464, 2.0)^T$
- B. $(-2.0, 3.464)^T$

[Answer 22](#)

Question 23

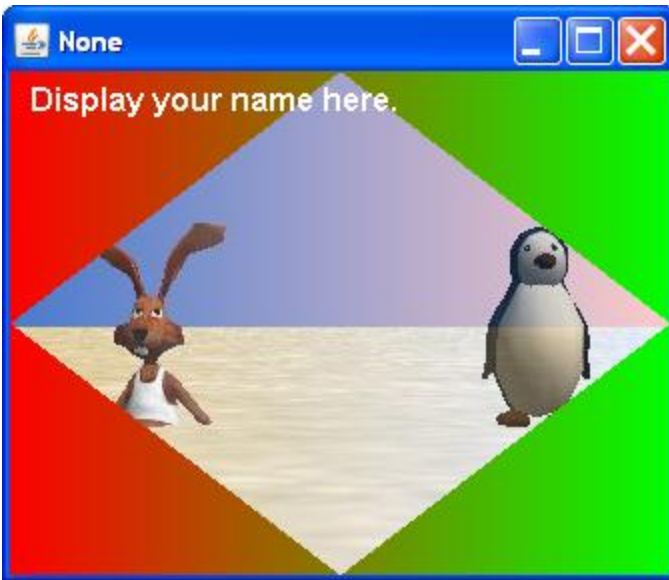
True or False: $2 \cdot \pi$ radians = 180 degrees

where the asterisk indicates multiplication and pi is the constant that begins with 3.14159

[Answer 23](#)

What is the meaning of the following two images?

These images were inserted here simply to insert some space between the questions and the answers to keep them from being visible on the screen at the same time.



This image was also inserted for the purpose of inserting space between the questions and the answers.



Answers

Answer 23

False. There are 2π radians in a full circle, so

2π radians = 360 degrees

[Back to Question 23](#)

Answer 22

A. (-3.464,2.0)T

Kjell, Chapter 5

[Back to Question 22](#)

Answer 21

D. -135 degrees

Kjell, Chapter 5

[Back to Question 21](#)

Answer 20

C. -45 degrees

Kjell, Chapter 5

[Back to Question 20](#)

Answer 19

True. Arc tan(z) means "find the angle that has a tangent of z." When you do this with a calculator, remember that the answer may be in radians or in

degrees. Some calculators give you the option of using use either format.

Kjell, Chapter 5

[Back to Question 19](#)

Answer 18

C. 315 degrees, or

D.-45 degrees

depending on the convention in use.

Kjell, Chapter 5

[Back to Question 18](#)

Answer 17

True

Kjell, Chapter 5

[Back to Question 17](#)

Answer 16

False. If v is a vector, then $-v$ is a vector pointing in the opposite direction.

Kjell, Chapter 4.

[Back to Question 16](#)

Answer 15

False. It is possible for the sum of the squared elements to be equal without the elements themselves being equal. In that case, the lengths are the same but the directions are not the same.

Kjell, Chapter 4.

[Back to Question 15](#)

Answer 14

True. Since corresponding elements must be equal, so corresponding squares must be equal, so the sum must be equal, so the length must be equal.

Kjell, Chapter 4

[Back to Question 14](#)

Answer 13

False. The length property of a vector must be zero or positive. It cannot be negative.

Kjell, Chapter 4.

[Back to Question 13](#)

Answer 12

True

Kjell, Chapter 4

[Back to Question 12](#)

Answer 11

C. 6

Kjell, Chapter 4

[Back to Question 11](#)

Answer 10

False. The correct statement follows where $|(x,y,z)^T|$ represents a three-dimensional vector:

$$|(x,y,z)^T| = \sqrt{x^2 + y^2 + z^2}$$

Kjell, Chapter 4

[Back to Question 10](#)

Answer 9

True

Kjell, Chapter 4

[Back to Question 9](#)

Answer 8

False. The following statement is true where the left angle bracket followed by the equal sign indicates "less than or equal." This is known as the "triangle inequality."

$|u + v|$ is less than or equal $|u| + |v|$

(Note that there is a problem with displaying the left angle bracket using my xhtml to cnxml converter program. That is why it is spelled out in the above statement.)

Kjell, Chapter 4

[Back to Question 8](#)

Answer 7

C. 5

Kjell, Chapter 4

[Back to Question 7](#)

Answer 6

C. 5

Kjell, Chapter 4

[Back to Question 6](#)

Answer 5

C. 10

Kjell, Chapter 4

[Back to Question 5](#)

Answer 4

C. 4

The length of the vector is 4 units.

Kjell, Chapter 4

[Back to Question 4](#)

Answer 3

False. The correct statement is:

When a 2D vector is aligned with the y axis, the column matrix that represents it has a non-zero value in the second element, and zero for the first element.

Kjell, Chapter 4

[Back to Question 3](#)

Answer 2

True. as in (3,0)^T

Kjell, Chapter 4

[Back to Question 2](#)

Answer 1

B. 3

The length of the vector is 3 units.

Kjell, Chapter 4

[Back to Question 1](#)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Game0130r: Review: Putting the Game-Math Library to Work
- File: Game0130r.htm
- Published: 09/22/13
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales

nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0135: Venturing into a 3D World

Learn how to update the game-math library to support 3D math, how to program the equations for projecting a 3D world onto a 2D plane, and how to add vectors in 3D. Also learn about scaling, translation, and rotation of a point in both 2D and 3D, about the rotation equations and how to implement them in both 2D and 3D, and much more.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview of coming attractions](#)
- [Preview of this module](#)
- [Discussion and sample code](#)
 - [The game-math library named GM01](#)
 - [The program named GM01test02](#)
 - [The program named GM01test01](#)
 - [The method named GM01.Point3D.draw](#)
 - [The method named GM01.Vector3D.draw](#)
 - [The method named GM01.Line3D.draw](#)
 - [The program named GM01test05](#)
 - [The program named GM01test06](#)
 - [The program named StringArt02](#)
 - [The program named StringArt03](#)
- [Documentation for the GM01 library.](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary.](#)
- [What's next?](#)
- [Miscellaneous](#)

- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)
 - [Exercise 3](#)
 - [Exercise 4](#)
 - [Exercise 5](#)
 - [Exercise 6](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In a previous module, I presented and explained four programs that used the game-math library named **GM2D04**. One of those programs taught you how to use the **addVectorToPoint** method of the **GM2D04.Point** class to translate a geometric object from one location in space to a different location in space.

Another program taught you how to do the same thing but in a possibly more efficient manner.

The third program taught you how to do rudimentary animation using the game-math library.

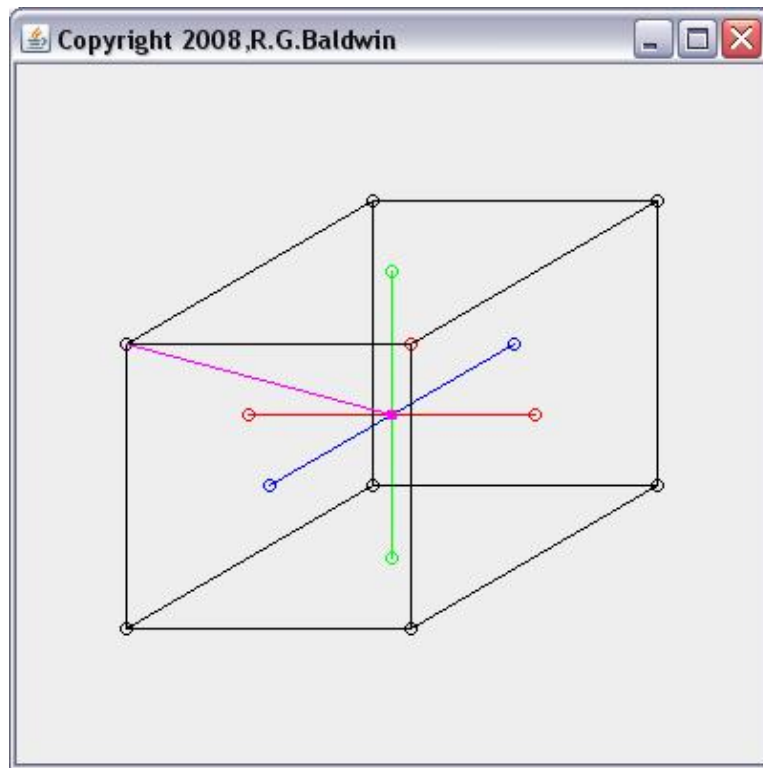
The fourth program taught you how to use methods of the game-math library to produce relatively complex drawings.

All of the programs were interactive in that they provide a GUI that allows the user to modify certain aspects of the behavior of the program.

What you will learn

The most important thing that you will learn in this module is how to update the game-math library to support 3D math and how to produce 3D images similar to that shown in [Figure 1](#).

Figure 1 A 3D image produced using the game-math library.



And a whole lot more...

You will learn much more than that however. Some highlights of the things you will learn are:

- How to program the equations for projecting a 3D world onto a 2D plane for display on a computer screen.
- How to cause the direction of the positive y-axis to be up the screen instead of down the screen.
- How to add vectors in 3D and how to confirm that the head-to-tail and parallelogram rules apply to 3D as well as to 2D.
- About scaling, translation, and rotation of a point in both 2D and 3D.
- About the rotation equations and how to implement them in both 2D and 3D.
- How to rotate a point around an anchor point other than the origin.
- About the right-hand rule.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). A 3D image produced using the game-math library.
- [Figure 2](#). Graphics output from the program named GM01test02.
- [Figure 3](#). Command-line output from the program named GM01test02.
- [Figure 4](#). Graphic output from the program named GM01test05.
- [Figure 5](#). Graphic output from the program named GM01test06.
- [Figure 6](#). Rotation equations for a point in 2D space.
- [Figure 7](#). Initial graphic output from the program named StringArt02.
- [Figure 8](#). Graphic output from the program named StringArt02 with Loops set to 3.
- [Figure 9](#). Rotation by 30 degrees around the origin.
- [Figure 10](#). Rotation around an anchor point that is not at the origin.
- [Figure 11](#). Rotation around a point further out in space.
- [Figure 12](#). The six 3D rotation equations.
- [Figure 13](#). Graphic output from the program named StringArt03 at startup.
- [Figure 14](#). Geometric object with 12 vertices, 4 loops, and no rotations.
- [Figure 15](#). Rotation around the z-axis only.
- [Figure 16](#). Rotation around the x-axis only.
- [Figure 17](#). Rotation around the y-axis only.
- [Figure 18](#). Rotation around all three axes with the anchor point at the origin.
- [Figure 19](#). Perform all three rotations with the anchor point away from the origin.
- [Figure 20](#). Screen output from Exercise 1.
- [Figure 21](#). Screen output from Exercise 2.
- [Figure 22](#). Screen output from Exercise 3.
- [Figure 23](#). Screen output from Exercise 4.
- [Figure 24](#). Screen output from Exercise 5.
- [Figure 25](#). Screen output from Exercise 6.

Listings

- [Listing 1](#). The static method named `convert3Dto2D`.
- [Listing 2](#). Transform equations for an oblique parallel projection from 3D to 2D.
- [Listing 3](#). Code in the constructor for the GUI class in the program named `GM01test02`.
- [Listing 4](#). The method named `GM01.Point3D.draw`.
- [Listing 5](#). The method named `GM01.Vector3D.draw`.
- [Listing 6](#). The `GM01.Line3D.draw` method.
- [Listing 7](#). Beginning of the `drawOffScreen` method in `GM01test01`.
- [Listing 8](#). Project eight points onto the 2D plane and draw them.
- [Listing 9](#). Draw twelve lines that connect the corners of the box.
- [Listing 10](#). Instantiate and draw a `GM01.Vector3D` object onto the 2D off-screen image.
- [Listing 11](#). The `setCoordinateFrame` method.
- [Listing 12](#). The `drawOffScreen` method of the program named `GM01test05`.
- [Listing 13](#). The game-math library method named `GM01.Point3D.scale`.
- [Listing 14](#). Beginning of the game-math library method named `GM01.Point2D.rotate`.
- [Listing 15](#). Translate the anchor point to the origin.
- [Listing 16](#). Rotate the translated `newPoint` object around the origin.
- [Listing 17](#). Translate the rotated `newPoint` object back to the anchor point.
- [Listing 18](#). Abbreviated listing of the `drawOffScreen` method.
- [Listing 19](#). Beginning of the method named `GM01.Point3D.rotate`.
- [Listing 20](#). Get the rotation angle values.
- [Listing 21](#). Rotate the `Point3D` object around the z-axis.
- [Listing 22](#). Rotate the `Point3D` object around the x-axis.
- [Listing 23](#). Rotate the `Point3D` object around the y-axis.
- [Listing 24](#). Translate the object back to the anchor point.
- [Listing 25](#). Interesting code from the `drawOffScreen` method.
- [Listing 26](#). Source code for the updated game-math library named `GM01`.
- [Listing 27](#). Source code for the program named `GM01test02`.
- [Listing 28](#). Source code for the program named `GM01test01`.
- [Listing 29](#). Source code for the program named `GM01test05`.
- [Listing 30](#). Source code for the program named `GM01test06`.
- [Listing 31](#). Source code for the program named `StringArt02`.
- [Listing 32](#). Source code for the program named `StringArt03`.

Preview of coming attractions

In case you are wondering where we are heading as we go down this path, I recommend that you skip ahead to the modules titled *Our First 3D Game Program* and *A First-Person Shooter Game* . Copy, compile and run the 3D game programs named **GM01Test08** and **Cannonball01** along with the required game-math libraries.

While the graphics and the story lines for those two game programs are rudimentary, the mathematics involved are significant. (*After all, this is a course in game math and not a course in game design or high-quality graphics.*)

Preview of this module

I will present and explain a significantly updated game-math library in this module.

I will also present and explain six different sample programs that show how to use the new features in the updated library.

By studying the library and the sample programs, you will learn

- how to update the game-math library to support 3D math,
- how to program the equations for projecting a 3D world onto a 2D plane,
- how to add vectors in 3D,
- about scaling, translation, and rotation of a point in both 2D and 3D,
- about the rotation equations and
- how to implement them in both 2D and 3D, and much more.

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this and previous modules.

Discussion and sample code

In this section, I will present and explain a significantly updated version of the game-math library named [GM01](#).

In addition, I will present and explain the following programs that use the update game-math library.

- [GM01test02](#)
- [GM01test01](#)
- [GM01test05](#)
- [GM01test06](#)
- [StringArt02](#)
- [StringArt03](#)

The game-math library named GM01

A complete listing of the updated game-math library named **GM01** is provided in [Listing 26](#) near the end of the module.

A major update to add 3D capability

This is a major update to the game-math library. This version updates the earlier version named **GM2D04** to a new version named simply **GM01** . The primary purpose of the update was to add 3D capability for all of the 2D features provided by the previous version. Because both 2D and 3D capabilities are now included in the library, it is no longer necessary to differentiate between the two in the name of the class. Therefore, this version is simply named **GM01** .

3D to 2D projections

Adding 3D capability was tedious, but not particularly difficult in most areas of the library. However, adding 3D capability entailed major complexity in one particular area: drawing the objects. It is difficult to draw a 3D object on a 2D screen and have the drawing appear to be 3D. This requires a projection process to project each point in the 3D object onto the correct location on a 2D screen. There are a variety of ways to do this. This 3D library uses an approach often referred to as an [oblique parallel projection](#) . You can Google that name to learn more about the technical details of the process.

Eliminating the y-axis confusion

In addition to adding 3D capability, this version of the game library also eliminates the confusion surrounding the fact that the default direction of the positive y-axis is down the screen instead of up the screen as viewers have become accustomed to. If you funnel all of your drawing tasks through the library and don't draw directly on the screen, you can program under the assumption that the positive direction of the y-axis is up.

The name of the library

The name **GM01** is an abbreviation for *GameMath01* . See the file named **GM2D01** from an earlier module for a general description of the game-math library. The library has been updated several times. This version updates the file named **GM2D04** .

Improving efficiency

In addition to the updates mentioned above, this update cleaned up some lingering areas of code inefficiency by using the simplest available method to draw on an off-screen image.

New static methods

Also, the following new static methods were added to the class named **GM01** . The first method in the following list deals with the problem of displaying a 3D image on a 2D screen. The last five methods in the list wrap around the standard graphics methods for the purpose of eliminating the issue of the direction of the positive Y-axis.

- GM01.convert3Dto2D
- GM01.translate
- GM01.drawLine
- GM01.fillOval
- GM01.drawOval
- GM01.fillRect

It will probably be necessary for me to add more wrapper methods to the library in future modules. With the exception of the first method in the above list, the coding of these methods was straightforward and explanations of that code are not warranted. Note, however, that it is the wrapper methods that resolve the issue regarding the direction of the positive y-axis. You will see

comments in this regard if you examine the source code for the wrapper methods in [Listing 27](#). I will explain the code for the method named **GM01.convert3Dto2D** shortly.

Other new methods

In addition to the new static methods listed above, a number of new methods were added to the existing static top-level 2D classes and also included in the new static top-level 3D classes. A list of those new methods follows:

- GM01.Vector2D.scale
- GM01.Vector2D.negate
- GM01.Point2D.clone
- GM01.Vector2D.normalize
- **GM01.Point2D.rotate**
- GM01.Point2D.scale
- GM01.Vector3D.scale
- GM01.Vector3D.negate
- GM01.Point3D.clone
- GM01.Vector3D.normalize
- **GM01.Point3D.rotate**
- GM01.Point3D.scale

With the exception of the two **rotate** methods, the coding of the methods in the above list was also straightforward and an explanation of that code is not warranted. You can view all of the new code in [Listing 26](#).

The two **rotate** methods are not straightforward at all. They are quite complicated (*particularly the 3D method*) and require quite a lot of background information to understand. I will dedicate a large portion of a future module to the task of rotating geometric objects in 2D and 3D worlds and will defer an explanation of the two **rotate** methods until that module.

The static method named GM01convert3Dto2D

Note first that this is a static method of the class named **GM01**. Among other things, this means that the method can be called simply by joining the name of the method to the name of the class. In other words, an object of the class named **GM01** is not necessary to make the method accessible.

A complete listing of the method is provided in [Listing 1](#).

Listing 1 . The static method named convert3Dto2D.

```
public static GM01.ColMatrix2D convert3Dto2D(
GM01.ColMatrix3D data){
    return new GM01.ColMatrix2D(
        data.getData(0) -
0.866*data.getData(2),
        data.getData(1) -
0.50*data.getData(2));
} //end convert3Dto2D
```

As you can see, the method is quite short, and once you know how it is required to [behave](#), coding the method is not difficult. The complexity comes in understanding the required behavior.

Note: Keeping 2D and 3D classes separate:

All of the static top-level 2D classes in the existing game-math library were renamed with a suffix of 2D to distinguish them from the new top-level 3D classes. All of the new top-level 3D class names have a 3D suffix.

The ColMatrix2D and ColMatrix3D classes

From the very beginning, the game-math library has contained a static top-level class named **ColMatrix** . In the updated version of the library, that class has been renamed **ColMatrix2D** .

Basically, the **ColMatrix2D** class provides a container for a pair of values of type **double** with appropriate methods for accessing those values. (*I explained the original **ColMatrix** class in detail in an earlier module.*) Similarly, the new **ColMatrix3D** class provides a container for three values of type **double** with appropriate methods for accessing those values.

Objects of the **ColMatrix2D** class are the fundamental building blocks for several of the other 2D classes in the library, and objects of the **ColMatrix3D** class are the fundamental building blocks for several of the other 3D classes in the library.

Behavior of the GM01.convert3Dto2D method

The **convert3Dto2D** method converts a **ColMatrix3D** object that represents a point in 3D space into a **ColMatrix2D** object that represents *the projection of that 3D point onto a 2D plane* .

The purpose of the method is to accept x, y, and z coordinate values describing a point in 3D space and to transform those values into a pair of coordinate values suitable for being displayed in two dimensions. The math that is implemented by this method to do the projection is described on a web page that seems to move around a bit but the last time I checked, it was located at <http://paulbourke.net/geometry/transformationprojection/> If you don't find it there, try <http://paulbourke.net/geometry/> In any event, a Google search should expose numerous pages that explain the math for projections of this type.

The transform equations

I won't attempt to justify or to explain the transform equations that are used to accomplish the projection in this module. Rather, I will simply use them as presented in the above resource. In the meantime, if you are interested in more information on the topic, you will find a wealth of information on 3D to 2D projections by performing a Google search for the topic.

The transform equations along with some of the assumptions that I made in the use of the equations are shown in [Listing 2](#).

Listing 2 . Transform equations for an oblique parallel projection from 3D to 2D.

The transform equations are:

$$\begin{aligned}x_{2d} &= x_{3d} + z_{3d} * \cos(\theta)/\tan(\alpha) \\ y_{2d} &= y_{3d} + z_{3d} * \sin(\theta)/\tan(\alpha);\end{aligned}$$

Let $\alpha = 45$ degrees and $\theta = 30$ degrees

Then: $\cos(\theta) = 0.866$

$\sin(\theta) = 0.5$

$\tan(\alpha) = 1;$

Note that the signs in the above equations depend on the assumed directions of the angles as well as

the assumed positive directions of the axes. The signs used in this method assume the following:

Positive x is to the right.

Positive y is up the screen.

Positive z is protruding out the front of the screen.

The viewing position is above the x axis and to the

right of the z - y plane.

The terms in the equations

In [Listing 2](#), the terms x_{2d} and y_{2d} refer to drawing coordinates on the 2D screen, while x_{3d} , y_{3d} , and z_{3d} refer to the coordinates of a point in 3D space. Obviously, \sin , \cos , and \tan refer to the sine, cosine, and tangent of angles named α and θ .

Output in two dimensions

These equations and the assumptions that I made in using them produce displays such as the one shown in [Figure 1](#). In that image, the red horizontal line is the x -axis with the positive direction to the right. The green vertical line

is the y-axis with the positive direction pointing up. The blue sloping line is the z-axis with the positive direction protruding from the screen towards and to the left of the viewer. The three axes intersect at the origin in 3D space.

(The sloping magenta line going from the corner of the box to the origin is a 3D vector. I will have more to say about the projections of 3D vectors later.)

No perspective in this projection

The sloping black lines in [Figure 1](#) represent the edges of a rectangular box projected onto the 2D screen. Note in particular that there is no perspective in this type of projection. In other words, lines that are parallel in the 3D space remain parallel in the projection of those lines onto the 2D screen. *(Hence the word parallel in the name oblique parallel projection.)* Objects don't appear to be smaller simply because they are further away from the viewer.

(The application of perspective would add another layer of complexity to the game math library. I will leave that as an exercise for the student to accomplish.)

Options involving the angles

I could have produced a somewhat different display by assuming different values for the angles named *alpha* and *theta* . However, the values chosen are commonly used values that produce reasonably good results, so I decided to use them. You may find it interesting to experiment with other values for one or both angles to see the results produced by those other values.

The proper algebraic signs

Note in particular that the proper signs for the equations in [Listing 2](#) depend on the assumed positive directions of the angles as well as the assumed positive directions of the axes. The signs used in the method make the assumptions shown in [Listing 2](#) . *(These assumptions will be particularly important in future modules where we will be rotating objects in 3D space.)*

The viewing position

Also as indicated in [Listing 2](#) and shown in [Figure 1](#) , the viewing position is above the x-axis and to the right of the z-y plane.

Typically, a game math library would provide the capability to modify the viewing position. That capability is not supported directly by this library. However, that capability can be approximated by the rotation capability discussed later. *(Another exercise for the student to accomplish.)*

The code in the method is straightforward

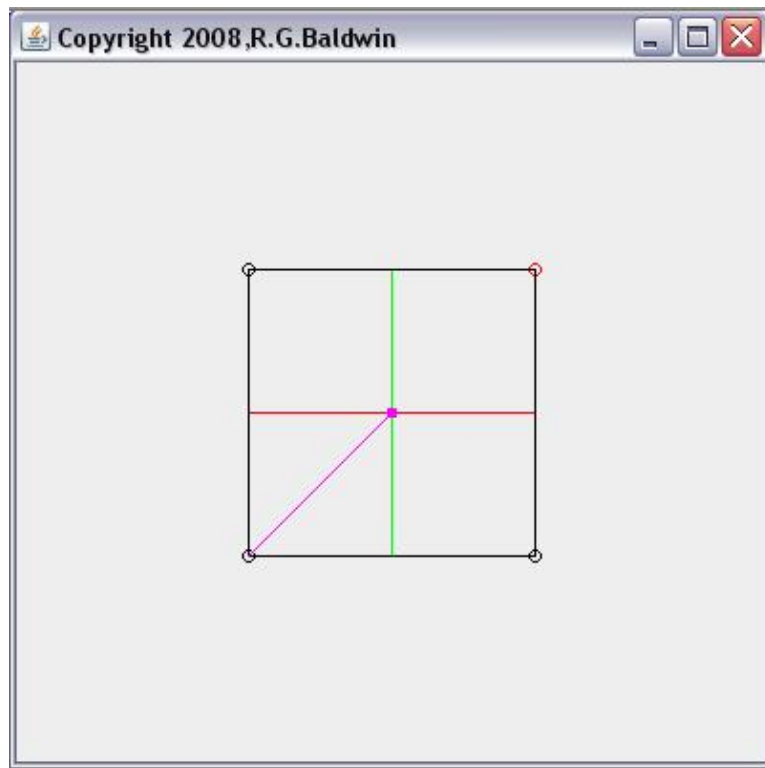
As I mentioned earlier, once you understand the requirements, the code for the method named **convert3Dto2D** (see [Listing 1](#)) is relatively straightforward. If you have studied the code in the previous modules in this series, the code in [Listing 1](#) shouldn't require further explanation.

The program named GM01test02

Because I made some changes to the existing 2D classes in the game-math library and added some new methods to some of the existing 2D classes, I felt the need to write a program that would step through and test the behavior of most of the 2D methods in the library. That was the purpose of the program named **GM01test02**.

This program produces both a graphic screen output and lots of text on the command-line screen. The graphic output is shown in [Figure 2](#).

Figure 2 Graphics output from the program named GM01test02.



Command-line output from the program named GM01test02

The command-line output is shown in [Figure 3](#).

Figure 3 . Command-line output from the program named GM01test02.

```
Test overridden toString of ColMatrix2D  
1.5,2.5
```

```
Test setData and getData of ColMatrix2D  
4.5  
5.5
```

```
Test equals method of ColMatrix2D  
false  
true
```

```
Test add method of ColMatrix2D  
9.0,11.0
```

```
Test subtract method of ColMatrix2D
```

0.0,0.0

Test toString method of Point2D

4.5,5.5

Test setData and getData of Point2D

1.1

2.2

Test getColMatrix method of Point2D

1.1,2.2

Test equals method of Point2D

false

true

Test getDisplacementVector method of Point2D

1.5,2.5

4.5,5.5

3.0,3.0

Test addVectorToPoint method of Point2D

1.5,2.5

7.0,9.0

Test toString method of Vector2D

1.5,2.5

Test setData and getData methods of Vector2D

4.5

5.5

Test getColMatrix method of Vector2D

4.5,5.5

Test equals method of Vector2D

false

true

Test add method of Vector2D

```
4.5,5.5  
-1.5,3.5  
3.0,9.0
```

Test getLength method of Vector2D

```
3.0,4.0  
5.0
```

Test toString method of Line2D

```
Tail = 1.5,2.5  
Head = 4.5,5.5
```

Test setTail, setHead, getTail,
and getHead methods of Line2D

```
4.5,5.5  
1.5,2.5
```

Not too exciting

Neither the graphics output in [Figure 2](#) nor the command-line output in [Figure 3](#) are terribly exciting, and neither will mean much to you unless you are willing to examine the code and compare the output with the code.

Source code for the program named GM01test02

A complete listing of the program named **GM01test02** is shown in [Listing 27](#) near the end of the module.

[Listing 3](#) shows two of the statements in the constructor for the **GUI** class in the program named **GM01test02** . The call to the method named **testUsingText** produces the command-line output shown in [Figure 3](#), while the call to the method named **drawOffScreen** produces the graphic output shown in [Figure 2](#) .

.

Listing 3 . Code in the constructor for the GUI class in the program named GM01test02.

```
//Perform tests using text.  
testUsingText();  
  
//Perform tests using graphics.  
drawOffScreen(g2D);
```

If you examine the source code for those two methods in [Listing 27](#), you will see that each of the methods contains calls to library methods for the purpose of confirming that the library methods behave as expected. The code in most of those methods is straightforward and should not require further explanation beyond the embedded comments in [Listing 27](#).

End of the discussion for the program named GM01test02

That concludes the discussion of the program named **GM01test02** . You will find a complete listing of this program in [Listing 27](#).

The program named GM01test01

Code in the program named **GM01test01** makes calls to library methods, which in turn make calls to the method named **convert3Dto2D** shown in [Listing 1](#). Because of the complexity of projecting 3D points onto a 2D screen for display, it will be useful to discuss the three library methods that make calls to the library method named **convert3Dto2D** . Those three methods are shown in the **following list** :

- GM01.Point3D.draw
- GM01.Vector3D.draw
- GM01.Line3D.draw

The method named GM01.Point3D.draw

Recall that a point simply represents a location in space and has no width, height, or depth. Therefore a point is not visible to the human eye. However, it is sometimes useful to draw a small circle around a point to mark its location for human consumption. That is the purpose of the **draw** method of the **GM01.Point3D** class, which is shown in its entirety in [Listing 4](#).

Listing 4 . The method named GM01.Point3D.draw.

```
public void draw(Graphics2D g2D){  
    //Get 2D projection coordinate values.  
    ColMatrix2D temp = convert3Dto2D(point);  
    drawOval(g2D, temp.getData(0)-3,  
             temp.getData(1)+3,  
             6,  
             6);  
} //end draw
```

Purpose of the method

This method draws a small circle around the location of a point in 3D space. The 3D location of the circle is projected onto the 2D plane of the specified graphics context for display later on a 2D screen. The code in [Listing 4](#) projects that location in 3D space onto the 2D plane by calling the static method named **convert3Dto2D** .

Behavior of the GM01.Point3D.draw method

The location of a point in 3D space, as represented by an object of the **GM01.Point3D** class, is actually stored in an object of the **GM01.ColMatrix3D** class. A reference to that **ColMatrix3D** object is stored

in the instance variable named **point** belonging to the **Point3D** object. This reference is passed as a parameter to the **convert3Dto2D** method in [Listing 4](#).

Recall from [Listing 1](#) that the **convert3Dto2D** method receives an incoming reference to an object of the class **GM01.ColMatrix3D** and returns a reference to an object of the class **GM01.ColMatrix2D**, which contains the horizontal and vertical components of the projection of the 3D point onto a 2D plane.

Using the returned **GM01.ColMatrix2D** object

[Listing 4](#) uses the horizontal and vertical components stored in the returned **ColMatrix2D** object to construct the proper parameters and call the wrapper method named **GM01.drawOval**. This wrapper method doesn't know that it is receiving parameters that were originally derived from an object in 3D space. The world of the wrapper method is confined to 2D space. The wrapper method named **GM01.drawOval** performs the following actions:

- casts the four incoming numeric parameters to type **int**,
- flips the sign on the vertical component to resolve the issue regarding the positive value for the y-axis, and
- passes the parameters to the **drawOval** method of the standard **Graphics** class to cause a small circle to be drawn at the correct location on the 2D off-screen image.

The small circle is not converted to an ellipse

Note that even though the location of the point in 3D space is projected onto the 2D plane, the shape of the small circle is not converted to an ellipse to reflect the 3D to 2D conversion nature of the operation. Thus, if the circle were large, it wouldn't necessarily look right. However, in this case, the only purpose of the circle is to mark the location of a point in 3D space. Therefore, I didn't consider the actual shape of the marker to be too important. [Figure 1](#) shows examples of circles marking points in 3D space at the corners of the box and at the ends of the axes.

The method named **GM01.Vector3D.draw**

The behavior described above for the method named **GM01.Point3D.draw** is somewhat indicative of the manner in which 3D geometric objects are projected onto a 2D plane and the manner in which the issue regarding the positive direction of the y-axis is resolved by the code in the updated game-math library.

The behavior of the three draw methods

Each of the three **draw** methods listed [earlier](#) needs the ability to project a **Point3D** object, a **Vector3D** object, or a **Line3D** object onto a 2D pane. In each case, the object to be drawn is built up using one or more **ColMatrix3D** objects as fundamental building blocks.

For the case of the **GM01.Point3D.draw** method discussed above, the **draw** method calls the **convert3Dto2D** method directly to convert the 3D coordinate values of the point to the 2D coordinate values required for the display.

A similar approach for the GM01.Vector3D.draw method

A similar approach is used for the **GM01.Vector3D.draw** method, which is shown in [Listing 5](#).

Listing 5 . The method named GM01.Vector3D.draw.

Listing 5 . The method named GM01.Vector3D.draw.

```
public void draw(Graphics2D g2D, GM01.Point3D
tail){

    //Get a 2D projection of the tail
    GM01.ColMatrix2D tail2D =
convert3Dto2D(tail.point);

    //Get the 3D location of the head
    GM01.ColMatrix3D head =

tail.point.add(this.getColMatrix());

    //Get a 2D projection of the head
    GM01.ColMatrix2D head2D =
convert3Dto2D(head);
    drawLine(g2D, tail2D.getData(0),
            tail2D.getData(1),
            head2D.getData(0),
            head2D.getData(1));

    //Draw a small filled circle to identify
the head.
    fillOval(g2D, head2D.getData(0)-3,
            head2D.getData(1)+3,
            6,
            6);

} //end draw
```

This method draws the 2D visual manifestation of a **GM01.Vector3D** object on the specified 2D graphics context. Recall that a vector has no location property. Therefore, it can be correctly drawn anywhere. The **GM01.Vector3D.draw** method requires the drawing location of the tail to be specified by a reference to a **GM01.Point3D** object received as an incoming parameter.

A small *filled* circle is drawn at the head of the vector as shown by the magenta filled circle at the origin in [Figure 1](#). (Note that the other circles in [Figure 1](#) are not filled.)

Two calls to the `convert3Dto2D` method

Two calls are made to the `convert3Dto2D` method in [Listing 5](#). The first call gets a 2D projection of the 3D location of the tail of the vector. The second call gets a 2D projection of the 3D location of the head of the vector. In both cases, the projected location in 2D space is returned as a reference to an object of the `GM01.ColMatrix2D` class.

Draw the vector on the specified drawing context

The reference returned by the first call to the `convert3Dto2D` method is used to call the static `GM01.drawLine` wrapper method to

- handle the issue of the positive direction of the y-axis, and
- draw a line on the 2D off-screen image representing the body of the vector as shown by the magenta line in [Figure 1](#).

The reference returned by the second call to the `convert3Dto2D` method is used to call the static `GM01.fillOval` wrapper method to

- handle the issue of the positive direction of the y-axis, and
- draw a small filled circle on the 2D off-screen image representing the head of the vector as shown by the magenta filled circle at the origin in [Figure 1](#).

The method named `GM01.Line3D.draw`

The `GM01.Line3D.draw` method is shown in [Listing 6](#).

Listing 6 . The GM01.Line3D.draw method.

```
public void draw(Graphics2D g2D){  
    //Get 2D projection coordinates.  
    GM01.ColMatrix2D tail =  
    convert3Dto2D(getTail().point);  
    GM01.ColMatrix2D head =  
    convert3Dto2D(getHead().point);  
  
    drawLine(g2D,tail.getData(0),  
            tail.getData(1),  
            head.getData(0),  
            head.getData(1));  
} //end draw
```

The code in [Listing 6](#) is so similar to the code in [Listing 5](#) that no further explanation should be required.

Now back to the program named GM01test01

A complete listing of the program named **GM01test01** is provided in [Listing 28](#) near the end of the module.

Because all of the 3D classes in the game-math library are new to this update, I felt the need to write a program that would step through and test the behavior of most of the 3D methods in the library. That was the purpose of the program named **GM01test01** .

Like the program named **GM01test02** discussed earlier, this program produces both a graphic screen output and lots of text on the command-line screen. The graphic output is shown in [Figure 1](#). I won't waste space printing the command-line output in this tutorial. If you want to see it, you can copy, compile, and run the program from [Listing 28](#) and produce that output yourself.

The graphic output

The graphic output shown in [Figure 1](#) is produced by the method named **drawOffScreen**, which begins in [Listing 7](#). I will briefly walk you through this method because everything in it is new. However, much of the code is very similar to 2D code that I have explained before.

Listing 7 . Beginning of the drawOffScreen method in GM01test01.

```
void drawOffScreen(Graphics2D g2D){  
    //Translate the origin on the off-screen  
    // image and draw a pair of orthogonal axes  
on it.  
    setCoordinateFrame(g2D);  
  
    //Define eight points that define the corners  
of  
    // a box in 3D that is centered on the  
origin.  
  
    GM01.Point3D[] points = new GM01.Point3D[8];  
    //Right side  
    points[0] =  
        new GM01.Point3D(new  
GM01.ColMatrix3D(75,75,75));  
    points[1] =  
        new GM01.Point3D(new  
GM01.ColMatrix3D(75,75,-75));  
    points[2] =  
        new GM01.Point3D(new  
GM01.ColMatrix3D(75,-75,-75));  
    points[3] =
```

Listing 7 . Beginning of the drawOffScreen method in GM01test01.

```
        new GM01.Point3D(new
GM01.ColMatrix3D(75, -75, 75));
        //Left side
        points[4] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, 75, 75));
        points[5] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, 75, -75));
        points[6] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, -75, -75));
        points[7] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, -75, 75));
```

After translating the origin to the center of the off-screen image, [Listing 7](#) instantiates eight **GM01Point3D** objects that define the corners of the 3D box shown in [Figure 1](#). References to the eight objects are stored in the elements of an array object referred to by the variable named **points** . Although this code uses three coordinate values instead of two coordinate values to instantiate the objects, the syntax should be very familiar to you by now.

Project eight points onto the 2D plane and draw them

[Listing 8](#) calls the **GM01.Point3D.draw** method eight times in succession to cause the locations of the eight points in 3D space to be projected onto the 2D off-screen image and to draw the points as small circles on that image. This is the first *draw* method from the [earlier list](#) that I explained above.

Listing 8 . Project eight points onto the 2D plane and draw them.

Listing 8 . Project eight points onto the 2D plane and draw them.

```
//Draw seven of the points in BLACK
g2D.setColor(Color.BLACK);
for(int cnt = 1;cnt < points.length;cnt++){
    points[cnt].draw(g2D);
} //end for loop

//Draw the right top front point in RED to
identify
// it.
g2D.setColor(Color.RED);
points[0].draw(g2D);
g2D.setColor(Color.BLACK);
```

Seven of the points are drawn in BLACK and one is drawn in RED. The eight small circles appear at the corners of the box in [Figure 1](#).

As indicated in the comments, the RED point is drawn at the right top front corner of the box to help you get the orientation of the box correct in your mind's eye.

Draw twelve lines that connect the corners of the box

[Listing 9](#) calls the **GM01.Line3D.draw** method twelve times in succession to project the lines that connect the corners of the 3D box onto the 2D off-screen image and to draw those lines on that image. This is the second *draw* method from the [earlier list](#) that I explained above.

Listing 9 . Draw twelve lines that connect the corners of the box.

Listing 9 . Draw twelve lines that connect the corners of the box.

```
//Draw lines that connect the points to
define the
// twelve edges of the box.
//Right side
new
GM01.Line3D(points[0],points[1]).draw(g2D);
new
GM01.Line3D(points[1],points[2]).draw(g2D);
new
GM01.Line3D(points[2],points[3]).draw(g2D);
new
GM01.Line3D(points[3],points[0]).draw(g2D);

//Left side
new
GM01.Line3D(points[4],points[5]).draw(g2D);
new
GM01.Line3D(points[5],points[6]).draw(g2D);
new
GM01.Line3D(points[6],points[7]).draw(g2D);
new
GM01.Line3D(points[7],points[4]).draw(g2D);

//Front
new
GM01.Line3D(points[0],points[4]).draw(g2D);
new
GM01.Line3D(points[3],points[7]).draw(g2D);

//Back
new
GM01.Line3D(points[1],points[5]).draw(g2D);
new
GM01.Line3D(points[2],points[6]).draw(g2D);
```


Instantiate and draw a GM01.Vector3D object onto the 2D off-screen image

[Listing 10](#) instantiates an object of the **GM01.Vector3D** class and calls the **draw** method of that class to draw the magenta vector shown in [Figure 1](#).

Listing 10 . Instantiate and draw a GM01.Vector3D object onto the 2D off-screen image.

```
//Instantiate a vector.
GM01.Vector3D vecA = new GM01.Vector3D(
                        new
GM01.ColMatrix3D(75, -75, -75));

//Draw the vector with its tail at the upper-
left
// corner of the box. The length and
direction of the
// vector will cause its head to be at the
origin.
g2D.setColor(Color.MAGENTA);
vecA.draw(g2D, points[4]);

} //end drawOffScreen
```

This is the third and final *draw* method from the [earlier list](#) that I explained above. The vector was drawn with its tail at the upper left corner of the box. The length and direction of the vector were such as to cause the head of the vector to be at the origin in 3D space.

Draw the 3D axes

The red, green, and blue 3D axes shown in [Figure 1](#) were produced by the call to the **setCoordinateFrame** method early in [Listing 7](#). The **setCoordinateFrame** method is shown in [Listing 11](#).

Listing 11 . The setCoordinateFrame method.

```
private void setCoordinateFrame(Graphics2D g2D)
{
    //Translate the origin to the center.
    GM01.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM01.Point3D pointA =
        new GM01.Point3D(new
GM01.ColMatrix3D(-75,0,0));
    GM01.Point3D pointB =
        new GM01.Point3D(new
GM01.ColMatrix3D(75,0,0));
    pointA.draw(g2D);
    pointB.draw(g2D);
    new GM01.Line3D(pointA, pointB).draw(g2D);

    //Draw y-axis in GREEN
    g2D.setColor(Color.GREEN);
    pointA =
        new GM01.Point3D(new
GM01.ColMatrix3D(0, -75,0));
    pointB =
        new GM01.Point3D(new
GM01.ColMatrix3D(0, 75,0));
```

Listing 11 . The setCoordinateFrame method.

```
pointA.draw(g2D);
pointB.draw(g2D);
new GM01.Line3D(pointA,pointB).draw(g2D);

//Draw z-axis in BLUE
g2D.setColor(Color.BLUE);
pointA =
    new GM01.Point3D(new
GM01.ColMatrix3D(0,0,-75));
pointB =
    new GM01.Point3D(new
GM01.ColMatrix3D(0,0,75));
pointA.draw(g2D);
pointB.draw(g2D);
new GM01.Line3D(pointA,pointB).draw(g2D);

}//end setCoordinateFrame method
```

This method is used to set the origin of the off-screen image. It also projects orthogonal 3D axes onto the 2D off-screen image and draws the projected axes on that image. The axes intersect at the origin in 3D space.

The lengths of the axes are set so as to match the interior dimensions of the box shown in [Figure 1](#) and points are drawn where the axes intersect the surfaces of the box. That was done to enhance the optical illusion of a 3D object on a 2D plane.

There is nothing in [Listing 11](#) that you haven't seen before, so further explanation should not be required.

End of discussion

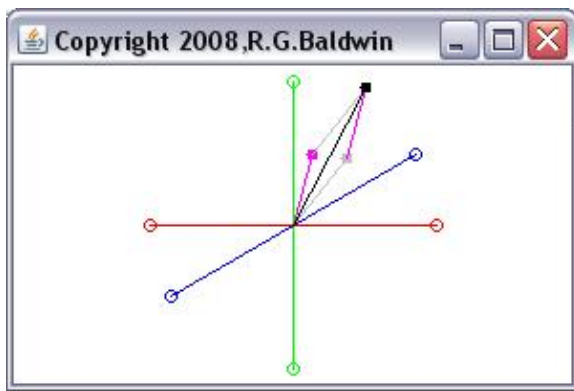
That concludes the discussion of the program named **GM01test01** . You can view the code that was not discussed here in [Listing 28](#) near the end of the module.

The program named GM01test05

In an earlier module, you learned how to add two or more vectors in 2D. This program will show you how to use the updated game-math library to add vectors in 3D. You also learned about the 2D parallelogram rule and the 2D head-to-tail rule in the earlier module. This program will illustrate that those rules also apply to the addition of 3D vectors.

[Figure 4](#) shows the graphic output produced by this program.

Figure 4 Graphic output from the program named GM01test05.



This output shows the addition of a magenta vector to a light gray vector to produce a black vector as the sum of the other two vectors. As you can see, both the parallelogram rule and the head-to-tail rule are illustrated by the graphic output in [Figure 4](#).

Very familiar code

Most of the code in this program will be very familiar to you by now. The new code is mostly contained in the method named **drawOffScreen**, which is shown in [Listing 12](#). A complete listing of this program is provided in [Listing 29](#) near the end of the module.

Listing 12 . The drawOffScreen method of the program named GM01test05.

```
//The purpose of this method is to illustrate
vector
// addition in 3D
void drawOffScreen(Graphics2D g2D){

    //Translate the origin on the off-screen
    image, draw a
    // pair of orthogonal axes on it that
    intersect at the
    // origin, and paint the background white.
    setCoordinateFrame(g2D);

    //Define two vectors that will be added.
    GM01.Vector3D vecA = new GM01.Vector3D(
        new
        GM01.ColMatrix3D(75,75,75));

    GM01.Vector3D vecB = new GM01.Vector3D(
        new
        GM01.ColMatrix3D(-15,10,-50));

    //Create a ref point at the origin for
    convenience.
    GM01.Point3D zeroPoint = new GM01.Point3D(
        new
        GM01.ColMatrix3D(0,0,0));

    //Draw vecA in MAGENTA with its tail at the
    origin.
    g2D.setColor(Color.MAGENTA);
    vecA.draw(g2D,zeroPoint);

    //Draw vecB in LIGHT_GRAY with its tail at
    the head
```

Listing 12 . The drawOffScreen method of the program named GM01test05.

```
// of vecA.
g2D.setColor(Color.LIGHT_GRAY);
GM01.Point3D temp =
    new
GM01.Point3D(vecA.getColMatrix());
vecB.draw(g2D,temp);

//Draw vecB in LIGHT_GRAY with its tail at
the origin.
vecB.draw(g2D,zeroPoint);

//Draw vecA in MAGENTA with its tail at the
head
// of vecB. This completes a trapezoid.
g2D.setColor(Color.MAGENTA);
vecA.draw(g2D,new
GM01.Point3D(vecB.getColMatrix()));

//Add the two vectors.
GM01.Vector3D sum = vecA.add(vecB);
//Draw sum in BLACK with its tail at the
origin.
g2D.setColor(Color.BLACK);
sum.draw(g2D,zeroPoint);

} //end drawOffScreen
```

At this point, you should not find any code in [Listing 12](#) that you don't understand. I have explained the code in [Listing 12](#) earlier in this module or in an earlier module, so I won't repeat that explanation here.

End of discussion .

That concludes the discussion of the program named **GM01test05** . You will find a complete listing of this program in [Listing 29](#).

The program named GM01test06

Scaling, translation, and rotation of a point

Three of the most common and important operations that you will encounter in game programming will be to scale, translate, and/or rotate a geometric object.

Since all geometric objects are composed of points that define the vertices, possibly embellished with lines, shading, lighting, etc., if you know how to scale, translate, and/or rotate a point, you also know how to scale, translate, and/or rotate the entire geometric object. You simply apply the required operation to all of the points that comprise the object and you will have applied the operation to the object as a whole.

However, there are some complex issues associated with hiding the back faces of a rotated object, which will be addressed in a future module under the general topic of *backface culling*. Backface culling is a process by which you prevent 3D objects from appearing to be transparent.

Note: Homogeneous Coordinates:

In a future module, we will learn about a mathematical trick commonly known as homogeneous coordinates that can simplify the scaling, translation, and rotation of a point. However, that will have to wait until we have a better grasp of matrix arithmetic.

In a previous module, you learned how to translate a geometric object. In this program, you will learn how to scale a geometric object. In the next two programs, you will learn how to rotate geometric objects.

The new scaling methods

The additions to the game-math library included the following two methods:

- GM01.Point2D.scale
- GM01.Point3D.scale

As the names imply, these two methods can be used to scale a point in either 2D or 3D.

The game-math library method named **GM01.Point3D.scale**

This method is shown in [Listing 13](#).

Listing 13 . The game-math library method named GM01.Point3D.scale.

```
public GM01.Point3D scale(GM01.ColMatrix3D
scale){
    return new GM01.Point3D(new ColMatrix3D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1),
                                getData(2) *
scale.getData(2)));
} //end scale
```

This method multiplies each coordinate value of the **Point3D** object on which the method is called by the corresponding values in an incoming **ColMatrix3D** object to produce and return a new **Point3D** object. This makes it possible to scale each coordinate value that defines the location in space by a different scale factor.

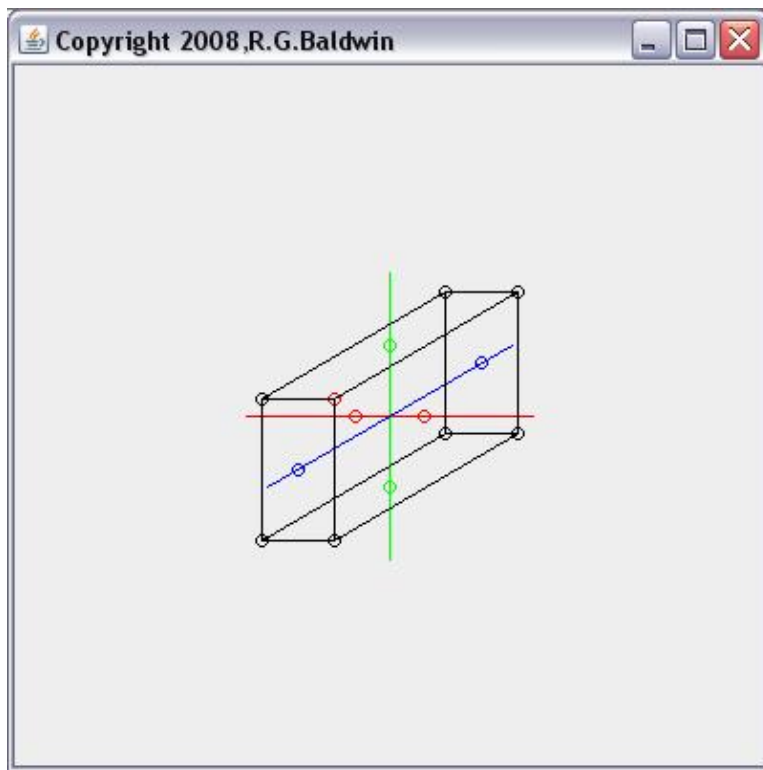
If a scale factor for a particular coordinate is less than 1.0 but greater than 0.0, the location of the new point will be closer to the origin along that axis than was the location of the original point. If a scale factor is greater than 1.0, the new point will be further from the origin along that axis. If the scale factor is

negative, the location of the new point will be on the other side of the origin along the same axis.

Graphic output from the program named GM01test06

The graphic output from the program named **GM01test06** is shown in [Figure 5](#). Compare this with the graphic output from the program named **GM01test01** shown in [Figure 1](#).

Figure 5 Graphic output from the program named GM01test06.



Modifications to the original program

The program named **GM01test06**, shown in [Listing 30](#), modifies the earlier program named **GM01test01** in three different ways to demonstrate the effects of scaling points in 3D.

1. Each **GM01Point3D** object that defines a corner of the box was scaled by a **ColMatrix3D** object containing the values 0.25, 0.5, and 0.75. Thus, the overall size of the box was reduced in comparison with the box in shown [Figure 1](#).

2. Each of the red, green, and blue points on the axes was scaled by 0.25, 0.5, or 0.75 respectively to cause the locations of those points to continue to be where the axes intersect the surface of the box.
3. The magenta vector shown in [Figure 1](#) was removed from the program because it didn't contribute to the objective of illustrating the scaling of a geometric object.

These modifications are so straightforward that no explanation of the actual code is justified. You can view the modifications in [Listing 30](#).

End of discussion .

That concludes the discussion of the program named **GM01test06** . You will find a complete listing of this program in [Listing 30](#).

The program named StringArt02

Things are about to get a lot more interesting. This program and the one following it both deal with the *rotation* of geometric objects, which is a relatively complex topic.

In this program, I will teach you how to use the new **GM01.Point2D.rotate** method to rotate objects in 2D space. In the next program, I will teach you how to use the new **GM01.Point3D.rotate** method to rotate objects in 3D space.

Rotation in 3D space is much more complicated than rotation in 2D space, so I will begin my explanation of this topic with the simpler of the two programs.

Equations for rotating an object in 2D space

Once again, we have some equations to deal with, and once we understand the equations, the required code is tedious, but not terribly difficult to write. The new game-math library method named **GM01.Point2D.rotate** , which begins in [Listing 14](#) , implements the required equations for rotating an object in 2D space.

Listing 14 . Beginning of the game-math library method named GM01.Point2D.rotate.

```
public GM01.Point2D rotate(GM01.Point2D
anchorPoint,
                        double angle){
    GM01.Point2D newPoint = this.clone();

    double tempX ;
    double tempY;
```

The purpose of the **GM01.Point2D.rotate** method is to rotate a point around a specified *anchor point* in the x-y plane. The location of the anchor point is passed in as a reference to an object of the class **GM01.Point2D** . The rotation angle is passed in as a **double** value in degrees with the positive angle of rotation being counter-clockwise.

Does not modify the original Point 2D object

This method does not modify the contents of the original **Point2D** object on which the method is called. Rather, it uses the contents of that object to instantiate, rotate, and return a new **Point2D** object, leaving the original object intact.

Equations for rotating an object in 2D space

By using the information on the [Spatial transformations](#) web page along with other information on the web, we can conclude that the equations required to rotate a point around the origin in 2D space are shown in [Figure 6](#).

Figure 6 . Rotation equations for a point in 2D space.

Figure 6 . Rotation equations for a point in 2D space.

$$\begin{aligned}x_2 &= x_1 \cdot \cos(\alpha) - y_1 \cdot \sin(\alpha) \\ y_2 &= x_1 \cdot \sin(\alpha) + y_1 \cdot \cos(\alpha)\end{aligned}$$

I won't attempt to derive or justify these equations. I will simply use them. If you need more information on the topic, simply Google *2D transforms* and you will probably find more information than you have the time to read.

In [Figure 6](#), the coordinates of the original point are given by **x1** and **y1**, and the coordinates of the rotated point are given by **x2** and **y2**. The angle **alpha** is a counter-clockwise angle around the origin.

Houston, we have a problem

We still have a problem however. The equations in [Figure 6](#) are for rotating a point around the origin. Our objective is to rotate a point around any arbitrary *anchor point* in 2D space.

We could probably modify the equations in [Figure 6](#) to accomplish this. However, there is another way, which is easier to implement. It can be shown that the same objective can be achieved by translating the anchor point to the origin, rotating the object around the origin, and then translating the rotated object back to the anchor point. Since we already know how to translate a point in 2D space, this is the approach that we will use.

You must be very careful

I do want to point out, however, that you really have to think about what you are doing when you rotate geometric objects, particularly when you combine rotation with translation. For example, rotating an object around the origin and then translating it does not produce the same result as translating the object and then rotating the translated object around the origin.

Clone the original Point2D object

[Listing 14](#) begins by calling the new **GM01.Point2D.clone** method to create a clone of the object on which the rotate method was called. The clone, referred to by **newPoint**, will be rotated and returned, thus preserving the original object.

Following that, [Listing 14](#) declares working variables that will be used later.

Incoming parameters

The **GM01.Point2D.rotate** method in [Listing 14](#) requires two incoming parameters. The first parameter is a reference to a **GM01.Point2D** object that specifies the anchor point around which the geometric object is to be rotated. The second parameter is the rotation angle in degrees, counter-clockwise around the origin.

Translate the anchor point to the origin

[Listing 15](#) gets a **Vector2D** object that represents the displacement vector from the origin to the anchor point.

Listing 15 . Translate the anchor point to the origin.

```
GM01.Vector2D tempVec =  
    new  
GM01.Vector2D(anchorPoint.getColMatrix());  
newPoint =  
  
newPoint.addVectorToPoint(tempVec.negate());
```

The negative of the displacement vector is used to translate the clone (**newPoint**) object, thus translating the anchor point to the origin.

Rotate the translated newPoint object around the origin

[Listing 16](#) implements the two rotation equations shown in [Figure 6](#) to rotate the translated **newPoint** object around the origin.

Listing 16 . Rotate the translated newPoint object around the origin.

```
tempX = newPoint.getData(0);
tempY = newPoint.getData(1);
newPoint.setData(new x coordinate
                  0,

tempX*Math.cos(angle*Math.PI/180) -
tempY*Math.sin(angle*Math.PI/180));
newPoint.setData(new y coordinate
                  1,

tempX*Math.sin(angle*Math.PI/180) +
tempY*Math.cos(angle*Math.PI/180));
```

Note that the rotation angle is converted from degrees to radians to make it compatible with the sin and cos functions from the standard Java Math library.

Translate the rotated newPoint object back to the anchor point

Finally, [Listing 17](#) uses the displacement vector that was created and saved earlier to translate the rotated **newPoint** object back to the anchor point.

Listing 17 . Translate the rotated newPoint object back to the anchor point.

```
        //Translate back to anchorPoint
        newPoint =
newPoint.addVectorToPoint(tempVec);

        return newPoint;

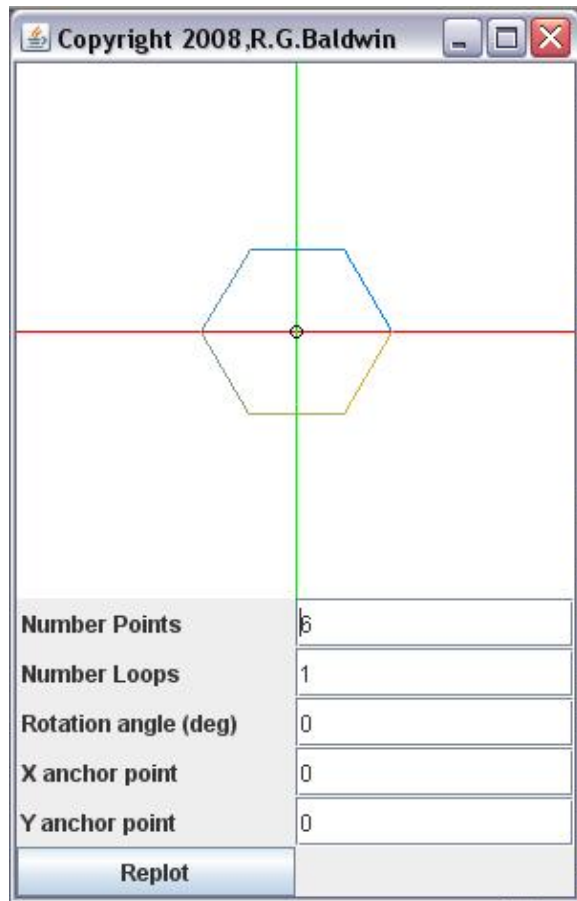
    }//end rotate
```

Then [Listing 17](#) returns a reference to the rotated **newPoint** object.

Now back to the program named StringArt02

This is a 2D version of a string art program that supports rotation in two dimensions. This program produces a 2D string art image by connecting various points that are equally spaced on the circumference of a circle as shown in [Figure 7](#).

Figure 7 Initial graphic output from the program named StringArt02.



Initial conditions

Initially, the circle is centered on the origin and there are six points on the circle connected by lines forming a hexagon. The lines that connect the points are different colors. The radius of the circle is 50 units. The points at the vertices of the hexagon are not drawn, but the lines that connect the vertices are drawn. The anchor point is drawn in black, resulting in the small black circle at the origin in [Figure 7](#).

A graphical user interface

A GUI is provided that allows the user to specify the following items and click a **Replot** button to cause the drawing to change:

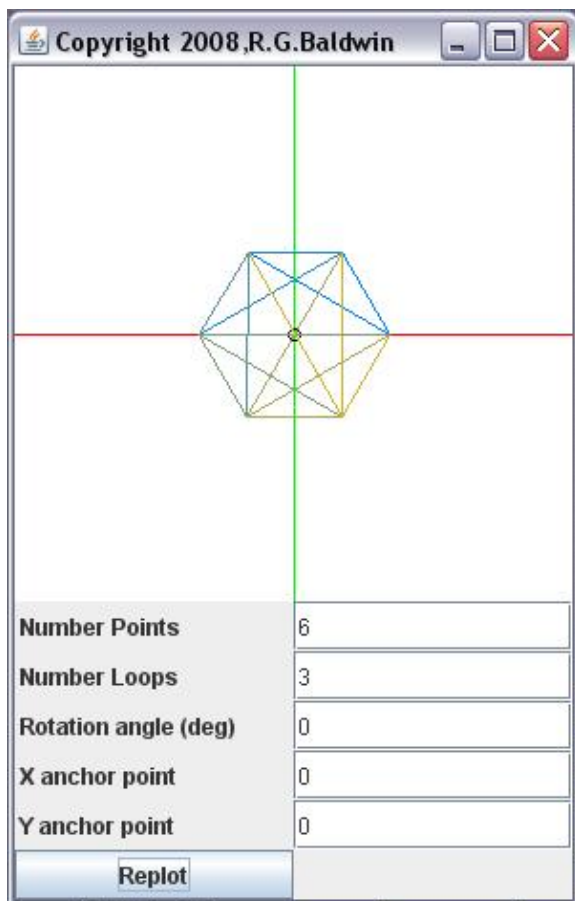
- Number Points
- Number Loops
- Rotation angle (deg)
- X anchor point

- Y anchor point

Changing the number of points causes the number of vertices that describe the geometric object to change. Changing the number of loops causes the number of lines that are drawn to connect the vertices to change.

For a value of 1, each vertex is connected to the one next to it. For a value of 2, additional lines are drawn connecting every other vertex. For a value of 3, additional lines are drawn connecting every third vertex as shown in [Figure 8](#).

Figure 8 Graphic output from the program named StringArt02 with Loops set to 3.

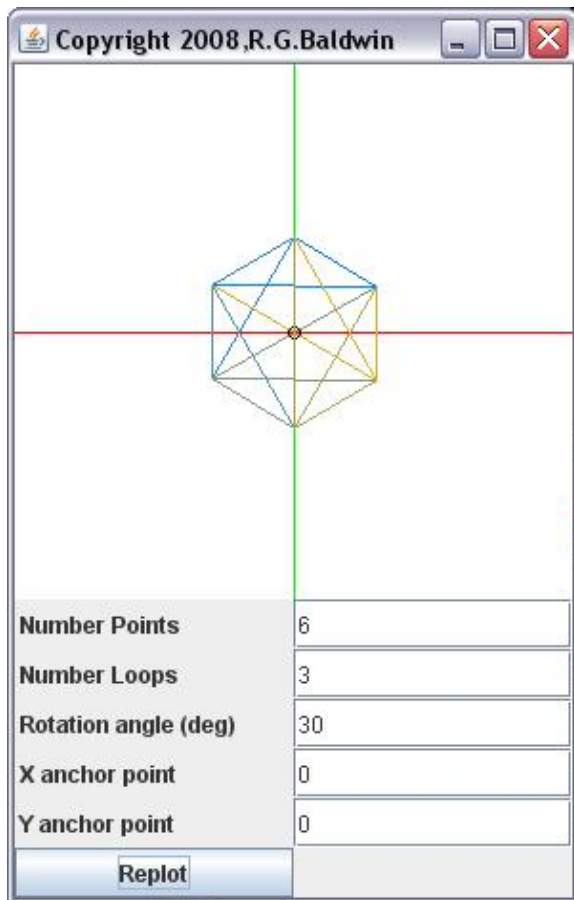


Rotating the geometric object around the origin

The geometric object can be rotated around an anchor point. Entering a non-zero value in the **Rotation** field causes the geometric object to be rotated by the specified angle around the anchor point.

The anchor point is initially located at the origin, but the location of the anchor point can be changed by the user. If the anchor point is at the origin, the geometric object is rotated around the origin as shown in [Figure 9](#). (Compare the colors and the locations of the vertices in [Figure 8](#) and [Figure 9](#) to discern the result of the rotation in [Figure 9](#).)

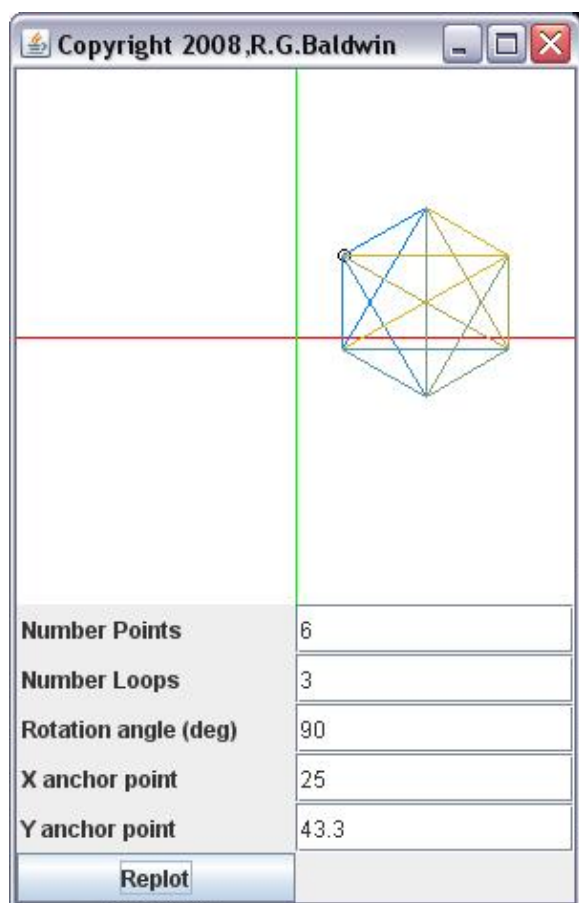
Figure 9 Rotation by 30 degrees around the origin.



Rotating the geometric object around an anchor point away from the origin

[Figure 10](#) shows the result of rotating the geometric object by 90 degrees around an anchor point that is not located at the origin.

Figure 10 Rotation around an anchor point that is not at the origin.



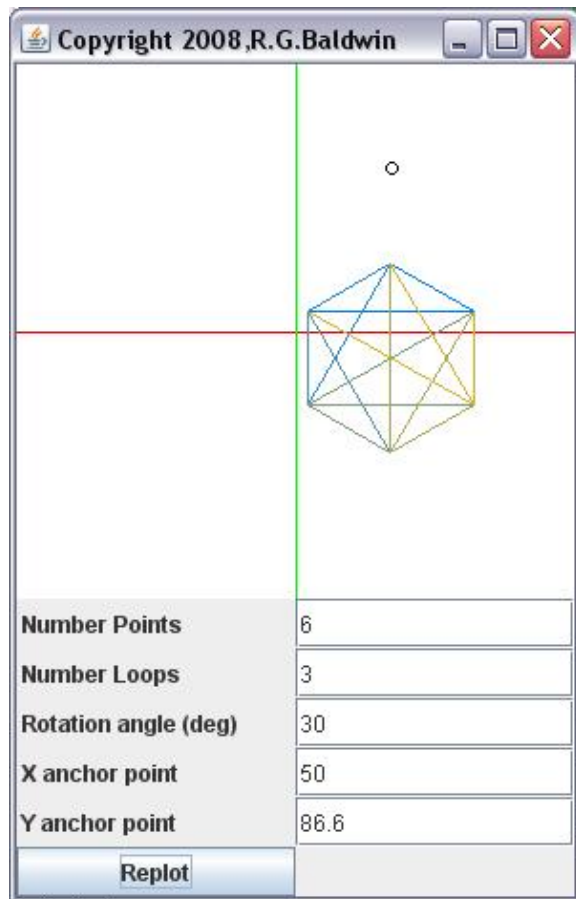
The rotation angle is specified in degrees with a positive angle being counter-clockwise. For [Figure 10](#), I purposely located the anchor point at the upper-right vertex in [Figure 8](#) and rotated the geometric object by 90 degrees around that anchor point. Compare [Figure 10](#) with [Figure 8](#) to see the result of rotating the geometric object around the anchor point.

Rotating around an anchor point further out in space

In [Figure 11](#), I moved the anchor point further out in space, but still on a line that runs through the origin and the upper-right vertex in [Figure 8](#). Then I rotated the geometric object by 30 degrees around the anchor point.

(Note that the rotation examples in these images are not cumulative. In other words, each rotation is relative to an un-rotated object at the origin.)

Figure 11 Rotation around a point further out in space.



By now, you should have been able to predict in advance what you would see when the program was run with these parameters.

Let's see some code

Given what you have already learned, the only interesting new code in this program is in the **drawOffScreen** method. An abbreviated listing of that method is shown in [Listing 18](#). A complete listing of the **StringArt02** program is provided in [Listing 31](#) near the end of the module.

Listing 18 . Abbreviated listing of the drawOffScreen method.

Listing 18 . Abbreviated listing of the drawOffScreen method.

```
void drawOffScreen(Graphics2D g2D){
    setCoordinateFrame(g2D,false);

    //Create a set of Point2D objects that specify
    // locations on the circumference of a circle
    that
    // is in the x-y plane with a radius of 50
    units. Save
    // references to the Point2D objects in an
    array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM01.Point2D(new
        GM01.ColMatrix2D(
        50*Math.cos((cnt*360/numberPoints)*Math.PI/180),
        50*Math.sin((cnt*360/numberPoints)*Math.PI/180)));

        //The following object is populated with the
        2D
        // coordinates of the point around which the
        //rotations will take place.
        GM01.Point2D anchorPoint = new GM01.Point2D(
            new GM01.ColMatrix2D(
            xAnchorPoint,yAnchorPoint));
        //Draw the anchorPoint in BLACK.
        g2D.setColor(Color.BLACK);
        anchorPoint.draw(g2D);

        //The following statement causes the
        rotation to be
        //performed.
        points[cnt] =
        points[cnt].rotate(anchorPoint,rotation);
```

Listing 18 . Abbreviated listing of the drawOffScreen method.

```
    }//end for loop  
  
    //Code deleted for brevity  
  
} //end drawOffScreen
```

Within the method named **drawOffScreen** , the only really interesting code is the statement that calls the **rotate** method of the game-math library on each **Point2D** object inside a **for** loop. Knowing what you do about the **rotate** method, you should have no problem understanding the code in [Listing 18](#).

End of the discussion

That concludes the discussion of the program named **StringArt02** . You will find a complete listing of this program in [Listing 31](#).

The program named StringArt03

I saved the best for last, or at least I saved the most difficult program until last. In this program, I will teach you how to use the new **GM01.Point3D.rotate** method to rotate objects in 3D space.

Not only is the code for doing rotations in 3D space much more complicated than the rotation code in 2D, it is also more difficult to examine the graphic output produced by rotating an object in 3D space and be certain that the program is working as it should. Therefore, we need to start this discussion with an explanation of the game-math library method named **GM01.Point3D.rotate** . Before we can get to that method, however, we must deal with the rotation equations for rotation of a point in 3D space.

The six 3D rotation equations

Unlike with 2D rotation where things were less complicated, we now have to deal with three coordinate values, three rotation angles, and six equations.

Using the [Spatial Transformations](#) webpage and other online material as well, we can conclude that our 3D rotation method must implement the six equations shown in [Figure 12](#).

Figure 12 . The six 3D rotation equations.

Let rotation angle around z-axis be zAngle
Let rotation angle around x-axis be xAngle
Let rotation angle around y-axis be yAngle

Rotation around the z-axis

$$\begin{aligned}x_2 &= x_1 \cdot \cos(z\text{Angle}) - y_1 \cdot \sin(z\text{Angle}) \\ y_2 &= x_1 \cdot \sin(z\text{Angle}) + y_1 \cdot \cos(z\text{Angle})\end{aligned}$$

Rotation around the x-axis

$$\begin{aligned}y_2 &= y_1 \cdot \cos(x\text{Angle}) - z_1 \cdot \sin(x\text{Angle}) \\ z_2 &= y_1 \cdot \sin(x\text{Angle}) + z_1 \cdot \cos(x\text{Angle})\end{aligned}$$

Rotation around the y-axis

$$\begin{aligned}x_2 &= x_1 \cdot \cos(y\text{Angle}) + z_1 \cdot \sin(y\text{Angle}) \\ z_2 &= -x_1 \cdot \sin(y\text{Angle}) + z_1 \cdot \cos(y\text{Angle})\end{aligned}$$

Where:

x1, y1, and z1 are coordinates of original point

x2, y2, and z2 are coordinates of rotated point

Also, as before, these six equations are only good for rotation around the origin, but our objective is to be able to rotate a point about any arbitrary anchor point in 3D space. Once again, we will use the trick of translating the anchor point to the origin, rotating the object around the origin, and then translating the object back to the anchor point.

Beginning of the method named GM01.Point3D.rotate

The method named **GM01.Point3D.rotate** begins in [Listing 19](#).

Listing 19 . Beginning of the method named GM01.Point3D.rotate.

```
public GM01.Point3D
```

The purpose of this method is to rotate a point around a specified anchor point in 3D space in the following order:

- Rotate around z - rotation in x-y plane.
- Rotate around x - rotation in y-z plane.
- Rotate around y - rotation in x-z plane.

Note: A useful upgrade:

A useful upgrade to the game-math library might be to write three separate rotation methods, each designed to rotate a Point3D object around only one of the three axes.

It is very important to understand that the order of the rotations is critical. You cannot change the order of rotations and expect to end up with the same results. This method is designed to allow you to rotate an object around all three axes with a single method call in the order given above. If you need to rotate your object in some different order, you should call the method up to three times in succession, rotating around only one axis with each call to the method.

Incoming parameters

The anchor point is passed in as a reference to an object of the **GM01.Point3D** class.

The rotation angles are passed in as double values in degrees (*based on the [right-hand rule](#)*) in the order given above, packaged in an object of the class **GM01.ColMatrix3D** .

*(Note that in this case, the **ColMatrix3D** object is simply a convenient container for the three **double** angle values and it has no significance from a matrix arithmetic viewpoint. Also pay attention to the order of the three values and the rotation axes associated with those values. See [Listing 20](#). It is z, x, y, and not x, y, z as you might assume.)*

The right-hand rule

The right-hand rule states that if you point the thumb of your right hand in the positive direction of an axis and curl your fingers to make a fist, the direction of positive rotation around that axis is given by the direction that your fingers will be pointing.

I will refer back to this rule later when describing rotations around various axes.

Original object is not modified

This method does not modify the contents of the **Point3D** object on which the method is called. Rather, it uses the contents of that object to instantiate, rotate, and return a new **Point3D** object.

Rotation around the anchor point

For simplicity, this method translates the anchor point to the origin, rotates around the origin, and then translates the object back to the anchor point.

Very familiar code

The code in [Listing 19](#) is very similar to the code that I explained earlier beginning in [Listing 14](#). Therefore, this code should not require an explanation beyond the embedded comments.

Get the rotation angle values

[Listing 20](#) extracts the rotation angle values from the **GM01.ColMatrix3D** object in which they are contained.

Listing 20 . Get the rotation angle values.

```
double zAngle = angles.getData(0);  
double xAngle = angles.getData(1);  
double yAngle = angles.getData(2);
```

Rotate the Point3D object around the z-axis

By this point in the execution of the method, the object has been translated to the origin using the negative of the anchor-point displacement vector. The object will be rotated around the origin and will then be translated back to the anchor point.

[Listing 21](#) implements the first two equations from [Figure 12](#) to rotate the **Point3D** object around the z-axis. By this, I mean that the object is rotated in a plane that is perpendicular to the z-axis modifying only x and y coordinate values from the object being rotated.

Listing 21 . Rotate the Point3D object around the z-axis.

Listing 21 . Rotate the Point3D object around the z-axis.

```
//Rotate around z-axis
tempX = newPoint.getData(0);
tempY = newPoint.getData(1);
newPoint.setData(//new x coordinate
                 0,

tempX*Math.cos(zAngle*Math.PI/180) -
tempY*Math.sin(zAngle*Math.PI/180));

                 newPoint.setData(//new y coordinate
                 1,

tempX*Math.sin(zAngle*Math.PI/180) +
tempY*Math.cos(zAngle*Math.PI/180));
```

This is the only rotation possibility in 2D rotation and the code in [Listing 21](#) is essentially the same as the code in [Listing 16](#) for 2D rotation.

Rotate the Point 3D object around the x-axis

Before translating the partially rotated object back to the anchor point, it must still be rotated around the x and y-axes. [Listing 22](#) implements the middle two equations in [Figure 12](#) to rotate the **Point3D** object in a plane that is perpendicular to the x-axis, modifying only the y and z coordinate values.

Listing 22 . Rotate the Point3D object around the x-axis.

Listing 22 . Rotate the Point3D object around the x-axis.

```
//Rotate around x-axis
tempY = newPoint.getData(1);
tempZ = newPoint.getData(2);
newPoint.setData(//new y coordinate
                 1,
tempY*Math.cos(xAngle*Math.PI/180) -
tempZ*Math.sin(xAngle*Math.PI/180));
newPoint.setData(//new z coordinate
                 2,
tempY*Math.sin(xAngle*Math.PI/180) +
tempZ*Math.cos(xAngle*Math.PI/180));
```

Rotate the Point3D object around the y-axis

[Listing 23](#) implements the last two equations in [Figure 12](#) to rotate the **Point3D** object in a plane that is perpendicular to the y-axis, modifying only the x and z coordinate values.

Listing 23 . Rotate the Point3D object around the y-axis.

Listing 23 . Rotate the Point3D object around the y-axis.

```
//Rotate around y-axis
tempX = newPoint.getData(0);
tempZ = newPoint.getData(2);
newPoint.setData(//new x coordinate
                 0,

tempX*Math.cos(yAngle*Math.PI/180) +
tempZ*Math.sin(yAngle*Math.PI/180));

newPoint.setData(//new z coordinate
                 2,
                 -
tempX*Math.sin(yAngle*Math.PI/180) +
tempZ*Math.cos(yAngle*Math.PI/180));
```

Translate the object back to the anchor point

[Listing 24](#) translates the rotated object back to the anchor point, thus completing the 3D rotation of a single **GM01.Point3D** object.

Listing 24 . Translate the object back to the anchor point.

Listing 24 . Translate the object back to the anchor point.

```
        //Translate back to anchorPoint
        newPoint =
newPoint.addVectorToPoint(tempVec);

        return newPoint;

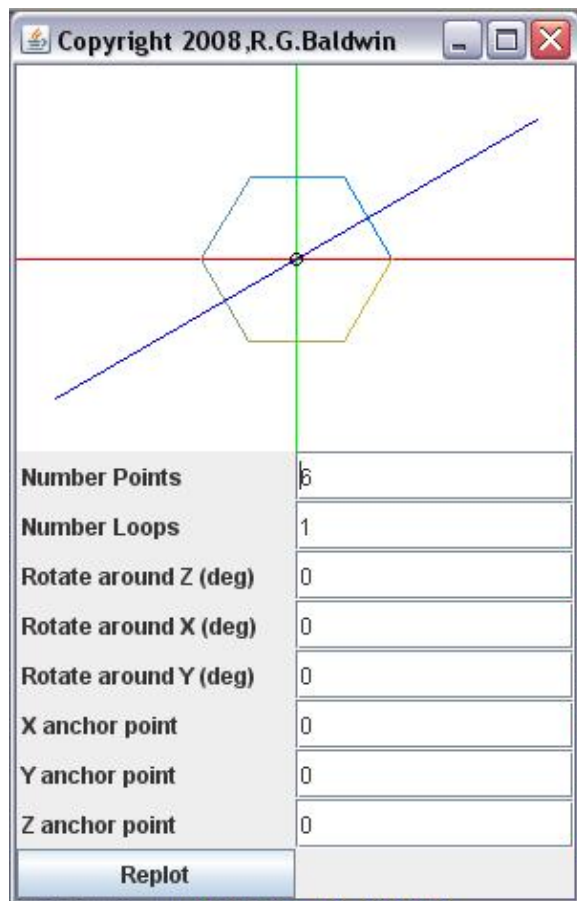
    }//end rotate
```

In order to rotate an entire 3D geometric object, such as the hexagon in [Figure 13](#), every point that comprises the geometric object must be rotated using the same set of rotation angles and the same anchor point.

Now back to the program named StringArt03

This is a 3D version of a string art program that demonstrates rotation in three dimensions. This program produces a 3D string-art image by connecting various points that are equally spaced on the circumference of a circle. Initially, the circle is on the x-y plane centered on the origin as shown in [Figure 13](#).

Figure 13 Graphic output from the program named StringArt03 at startup.



At startup, there are six points (*vertices*) on the circle connected by lines forming a hexagon. The lines that connect the points are different colors. The radius of the circle is 50 units. The points at the vertices of the hexagon are not drawn, but the lines that connect the vertices are drawn.

You may have noticed that the startup graphic output in [Figure 13](#) looks a lot like the startup graphic output of the 2D program in [Figure 7](#). There is a significant difference however. [Figure 7](#) shows only two orthogonal axes whereas [Figure 13](#) shows three orthogonal axes using oblique parallel projection to transform the 3D image to a 2D display plane.

A graphical user interface

A GUI is provided that allows the user to specify the following items and click a **Replot** button to cause the drawing to change:

- Number Points
- Number Loops

- Rotate around Z (deg)
- Rotate around X (deg)
- Rotate around Y (deg)
- X Anchor point
- Y Anchor point
- Z Anchor point

Again, the 3D GUI in [Figure 13](#) looks similar to the 2D GUI in [Figure 7](#). The big difference is that the 2D GUI in [Figure 7](#) allows only for rotation around one axis, and only two coordinate values can be specified for the location of the anchor point.

As before, changing the number of points causes the number of vertices that describe the geometric object to change. Changing the number of loops causes the number of lines that are drawn to connect the vertices to change.

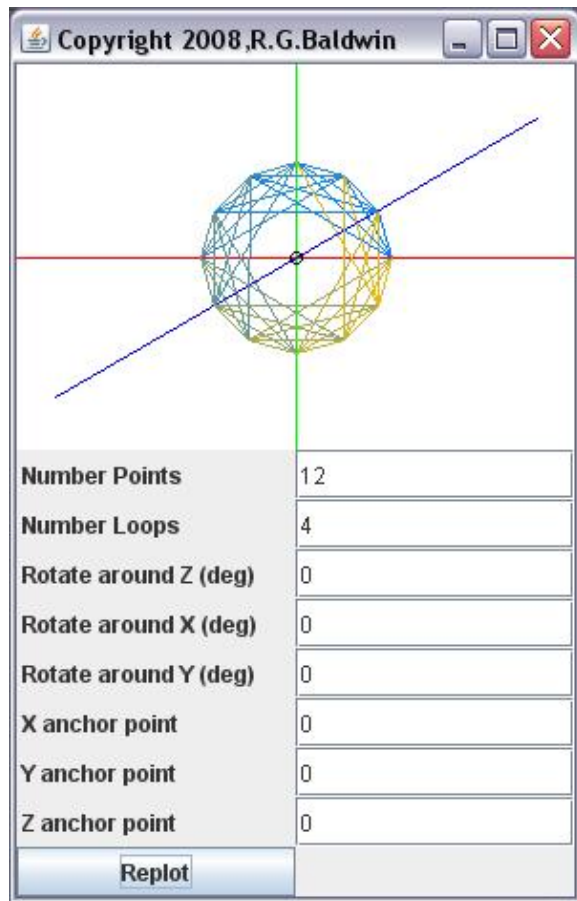
The geometric object can be rotated in any or all of three dimensions around an anchor point. Entering a non-zero value in one or more of the **Rotate** fields causes the object to be rotated by the specified angle or angles around the anchor point.

The anchor point is initially specified to be at the origin, but the location of the anchor point can be changed by the user. If the anchor point is at the origin, the image is rotated around the origin.

Geometric object with 12 vertices, 4 loops, and no rotations

As a baseline case, [Figure 14](#) shows the string-art geometric object with 12 vertices, 4 loops, and no rotations. At this point, the geometric object is an infinitely thin disk in the x-y plane centered on the origin. Note the break in color between yellow and blue that occurs where the circle crosses the positive x-axis.

Figure 14 Geometric object with 12 vertices, 4 loops, and no rotations.



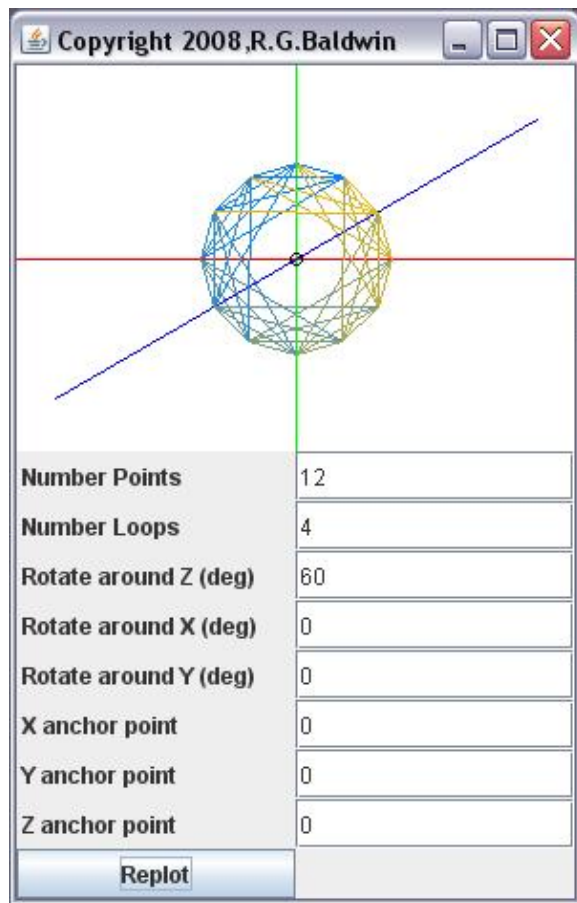
The rotation angle must be specified in degrees with a positive angle being given by the [right-hand rule](#) as applied to the axis around which the image is being rotated.

Rotation around one axis only

[Figure 15](#), [Figure 16](#), and [Figure 17](#) show the results of rotating the object around only one axis at a time with the anchor point at the origin.

[Figure 15](#) shows the result of rotating the object around the z-axis only by an angle of 60 degrees.

Figure 15 Rotation around the z-axis only.

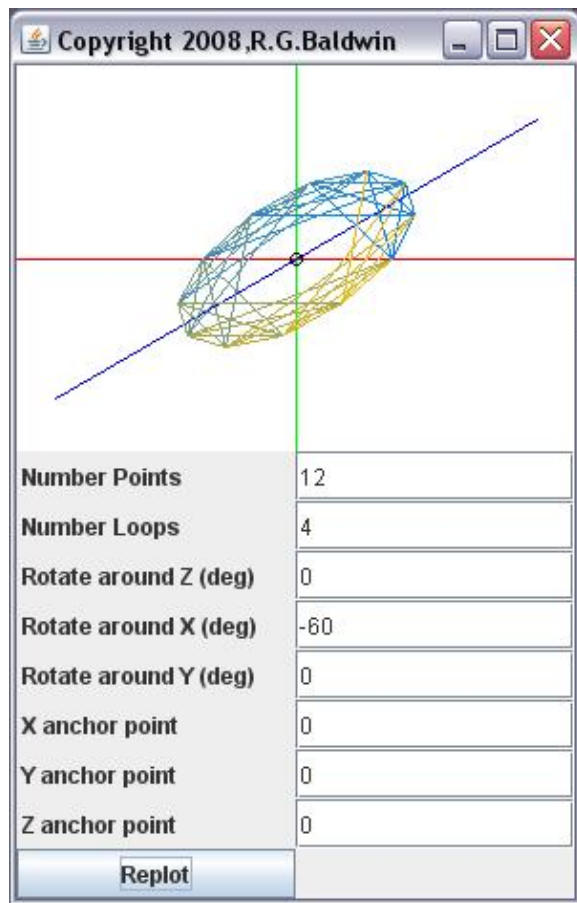


This results in the object still being in the x-y plane, but it has been rotated counter-clockwise by 60 degrees. Compare [Figure 15](#) with [Figure 14](#) and note how the color break between yellow and blue has moved around to be near the intersection of the circle and the positive y-axis.

Rotation around the x-axis only

[Figure 16](#) shows the result of rotating the object around only the x-axis with a rotation angle of -60 degrees.

Figure 16 Rotation around the x-axis only.

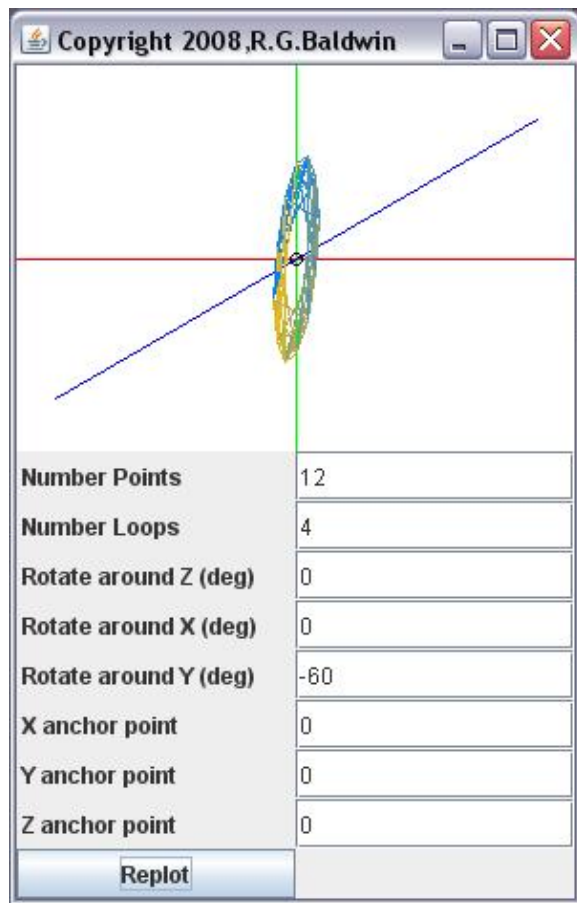


The object is still a disk, but that disk is no longer in the x-y plane. Instead, it has been tilted so that it is now closer to the x-z plane than to the x-y plane. Unfortunately, the oblique parallel projection does not make it practical to do any quantitative measurements on the image.

Rotation around the y-axis only

[Figure 17](#) shows the result of rotating the object around only the y-axis with a rotation angle of -60 degrees.

Figure 17 Rotation around the y-axis only.



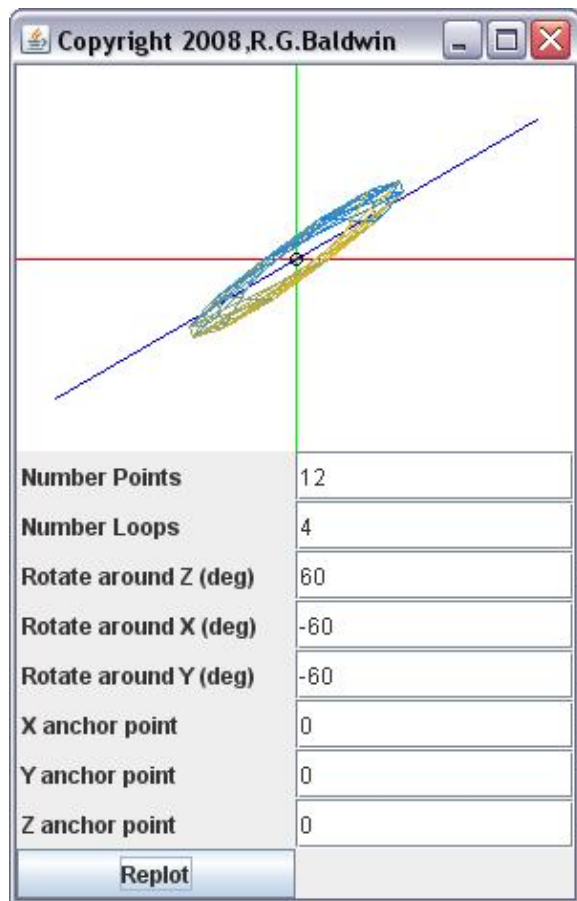
I will let you interpret what you see there.

Rotation around all three axes with the anchor point at the origin

When more than one rotation angle has a non-zero value, the rotational effects are cumulative. The object is first rotated around the anchor point in a direction consistent with rotation around the z-axis (*rotation in the x-y plane*). Then that rotated object is rotated in a direction consistent with rotation around the x-axis (*rotation in the y-z plane*). Finally, the previously rotated object is rotated in a direction consistent with rotation around the y-axis (*rotation in the x-z plane*). It is important to note, however, that the actual rotation is around the anchor point and not around the origin unless the anchor point is at the origin.

[Figure 18](#) shows the result of applying all three of the rotations described above with the anchor point at the origin.

Figure 18 Rotation around all three axes with the anchor point at the origin.

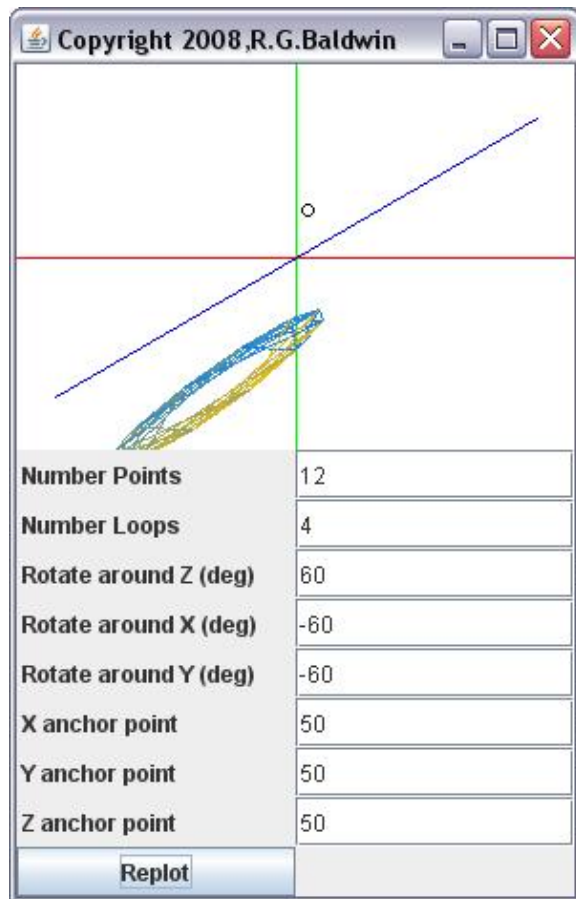


Once again, I will let you interpret what you see there.

Perform all three rotations with the anchor point away from the origin

[Figure 19](#) performs the same three rotations as were performed in [Figure 18](#). However, in [Figure 19](#), the anchor point was at a location defined by the coordinate values (50,50,50).

Figure 19 Perform all three rotations with the anchor point away from the origin.



At the risk of being boring, I will state once again that I will let you interpret what you see there.

Will animate the process later

For me, at least, it isn't easy to visualize the process of rotating around an arbitrary anchor point in 3D. In a future module, I will animate the rotation process and run it in slow motion so that you can see the progress of each individual rotation from the beginning until the point where all three rotations are complete. Hopefully, that will make it easier to visualize rotation around an arbitrary anchor point in 3D. As a bonus, it will also give you some experience in using the game-math library for a non-trivial animation project.

Let's see some code

As was the case with the previous program, given what you have already learned, the only interesting new code in this program is in the **drawOffScreen** method. Furthermore, only a small portion of the code in that method is new

and interesting. [Listing 25](#) contains some code that was extracted from the **drawOffScreen** method.

A complete listing of the program named **StringArt03** is provided in [Listing 32](#).

Listing 25 . Interesting code from the drawOffScreen method.

```
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = new GM01.Point3D(
            new GM01.ColMatrix3D(

50*Math.cos((cnt*360/numberPoints)*Math.PI/180),

50*Math.sin((cnt*360/numberPoints)*Math.PI/180),
            0.0));

        //Populate a ColMatrix3D object with
rotation values
        GM01.ColMatrix3D angles = new
GM01.ColMatrix3D(

zRotation,xRotation,yRotation);

        //Populate a Point3D object with anchor
point
        // coordinates.
        GM01.Point3D anchorPoint = new
GM01.Point3D(
            new GM01.ColMatrix3D(

xAnchorPoint,yAnchorPoint,zAnchorPoint));
```

Listing 25 . Interesting code from the drawOffScreen method.

```
//Draw the anchorPoint in BLACK.
g2D.setColor(Color.BLACK);
anchorPoint.draw(g2D);

//The following statement causes the
rotation to be
//performed.
points[cnt] =

points[cnt].rotate(anchorPoint,angles);

} //end for loop
```

The only really interesting code in [Listing 25](#) is the statement that calls the **rotate** method of the game-math library on each **Point3D** object inside a **for** loop near the bottom of the listing. Knowing what you do about the **rotate** method, you should have no problem understanding the code in [Listing 25](#).

End of the discussion

That concludes the discussion of the program named **StringArt03** . You will find a complete listing of this program in [Listing 32](#).

Documentation for the GM01 library

Click [here](#) to download a zip file containing standard javadoc documentation for the library named **GM01** . Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text (see [Listing 26](#) and the explanations given above) , the documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter6 - Scaling and Unit Vectors* .

The homework assignment for the next module is to continue studying the same material.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 26](#) through [Listing 32](#). Compile the code and execute it in conjunction with the game-math library provided in [Listing 26](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

The most important thing that you learned in this module was how to update the game-math library to support 3D math and how to produce 3D images similar to that shown in [Figure 1](#). You learned much more than that however. Some highlights of the things you learned are:

- You learned how to program the equations for projecting a 3D world onto a 2D plane for display on a computer screen.
- You learned how to cause the direction of the positive y-axis to be up the screen instead of down the screen.
- You learned how to add vectors in 3D and confirmed that the head-to-tail and parallelogram rules apply to 3D as well as to 2D.

- You learned about scaling, translation, and rotation of a point in both 2D and 3D.
- You learned about the rotation equations and how to implement them in both 2D and 3D.
- You learned how to rotate a point around an anchor point other than the origin.
- You learned about the right-hand rule.

What's next?

In the next module in this collection, you will learn how to write your first interactive 3D game using the game-math library. You will also learn how to write a Java program that simulates flocking behavior such as that exhibited by birds and fish and how to incorporate that behavior into a game.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0135: Venturing into a 3D World
- File: Game135.htm
- Published: 10/18/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 26](#) through [Listing 32](#) below.

Listing 26 . Source code for the updated game-math library named GM01.

```
/*GM01.java  
Copyright 2008, R.G.Baldwin  
Revised 02/24/08
```

This is a major upgrade to the game-math library.
This
version upgrades the version named GM2D04 to a new
version
named simply GM01.

The primary purpose of the upgrade was to add 3D
capability for all of the 2D features provided by the
previous version. Because both 2D and 3D capabilities
are included, it is no longer necessary to
differentiate
between the two in the name of the class. Therefore,
this
version is named GM01.

Adding 3D capability entailed major complexity in one
particular area: drawing the objects. It is difficult

to
draw a 3D object on a 2D screen. This requires a
projection process to project each point in the 3D
object
onto the correct location on a 2D plane. There are a
variety of ways to do this. This 3D library uses an
approach often referred to as an oblique parallel
projection. See the following URL for technical
information on the projection process:

[http://local.wasp.uwa.edu.au/~pbourke/geometry/
classification/](http://local.wasp.uwa.edu.au/~pbourke/geometry/classification/)

In addition to adding 3D capability, this version
also
eliminates the confusion surrounding the fact that
the
default direction of the positive y-axis is going
down
the screen instead of up the screen as viewers have
become
accustomed to. When you use this library, you can
program
under the assumption that the positive direction of
the
y-axis is up the screen, provided you funnel all of
your
drawing tasks through the library and don't draw
directly on the screen.

The name GMnn is an abbreviation for GameMathnn.

See the file named GM2D01.java for a general
description
of the game-math library. The library has been
updated
several times. This file is an update of GM2D04.

In addition to the updates mentioned above, this update cleaned up some lingering areas of code inefficiency, using the simplest available method to draw on an off-screen image. In addition, the following new methods were added:

The following methods are new static methods of the class named GM01. The first method in the list deals with the problem of displaying a 3D image on a 3D screen.

The last five methods in the list wrap the standard graphics methods for the purpose of eliminating the issue of the direction of the positive Y-axis.

GM01.convert3Dto2D
GM01.translate
GM01.drawLine
GM01.fillOval
GM01.drawOval
GM01.fillRect

The following methods are new instance methods of the indicated static top-level classes belonging to the class named GM01.

GM01.Vector2D.scale
GM01.Vector2D.negate
GM01.Point2D.clone
GM01.Vector2D.normalize
GM01.Point2D.rotate
GM01.Point2D.scale

GM01.Vector3D.scale

```
GM01.Vector3D.negate
GM01.Point3D.clone
GM01.Vector3D.normalize
GM01.Point3D.rotate
GM01.Point3D.scale
```

Tested using JDK 1.6 under WinXP.

```
*****
```

```
****/
```

```
import java.awt.geom.*;
import java.awt.*;
```

```
public class GM01{
```

```
    //-----
    ---//
```

```
    //This method converts a ColMatrix3D object to a
    // ColMatrix2D object. The purpose is to accept
    // x, y, and z coordinate values and transform
    those
    // values into a pair of coordinate values suitable
    for
```

```
    // display in two dimensions.
    //See
```

```
http://local.wasp.uwa.edu.au/~pbourke/geometry/
    // classification/ for technical background on the
    // transform from 3D to 2D.
```

```
    //The transform equations are:
    //  $x_{2d} = x_{3d} + z_{3d} * \cos(\theta)/\tan(\alpha)$ 
    //  $y_{2d} = y_{3d} + z_{3d} * \sin(\theta)/\tan(\alpha)$ ;
    //Let  $\theta = 30$  degrees and  $\alpha = 45$  degrees
    //Then: $\cos(\theta) = 0.866$ 
    //       $\sin(\theta) = 0.5$ 
    //       $\tan(\alpha) = 1$ ;
    //Note that the signs in the above equations depend
    // on the assumed directions of the angles as well
    as
```

```
    // the assumed positive directions of the axes. The
    // signs used in this method assume the following:
```

```

    //    Positive x is to the right.
    //    Positive y is up the screen.
    //    Positive z is protruding out the front of the
    //        screen.
    //    The viewing position is above the x axis and
to the
    //        right of the z-y plane.
    public static GM01.ColMatrix2D convert3Dto2D(
                                                GM01.ColMatrix3D
data){
    return new GM01.ColMatrix2D(
        data.getData(0) -
0.866*data.getData(2),
        data.getData(1) -
0.50*data.getData(2));
    }//end convert3Dto2D
    //-----
    ---//

    //This method wraps around the translate method of
the
    // Graphics2D class. The purpose is to cause the
    // positive direction for the y-axis to be up the
screen
    // instead of down the screen. When you use this
method,
    // you should program as though the positive
direction
    // for the y-axis is up.
    public static void translate(Graphics2D g2D,
                                double xOffset,
                                double yOffset){
        //Flip the sign on the y-coordinate to change the
        // direction of the positive y-axis to go up the
        // screen.
        g2D.translate(xOffset, -yOffset);
    }//end translate
    //-----
    ---//

```

```

    //This method wraps around the drawLine method of
the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive direction
for
    // the y-axis is up.
    public static void drawLine(Graphics2D g2D,
                                double x1,
                                double y1,
                                double x2,
                                double y2){
        //Flip the sign on the y-coordinate value.
        g2D.drawLine((int)x1, -(int)y1, (int)x2, -(int)y2);
    }//end drawLine
    //-----
    ---//

```

```

    //This method wraps around the fillOval method of
the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive direction
for
    // the y-axis is up.
    public static void fillOval(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){

```



```

        //Flip the sign on the y-coordinate value.
        g2D.fillOval((int)x, -(int)y, (int)width,
(int)height);
    }//end fillOval
    //-----
    ---//

```

```

    //This method wraps around the drawOval method of
the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive direction
for
    // the y-axis is up.
    public static void drawOval(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.drawOval((int)x, -(int)y, (int)width,
(int)height);
    }//end drawOval
    //-----
    ---//

```

```

    //This method wraps around the fillRect method of
the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive direction

```

```

for
    // the y-axis is up.
    public static void fillRect(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.fillRect((int)x,-(int)y,(int)width,
(int)height);
    }//end fillRect
    //-----
---//

```

```

    //An object of this class represents a 2D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix2D{
        double[] data = new double[2];

        public ColMatrix2D(double data0,double data1){
            data[0] = data0;
            data[1] = data1;
        }//end constructor
        //-----
---//

```

```

        //Overridden toString method.
        public String toString(){
            return data[0] + "," + data[1];
        }//end overridden toString method
        //-----
---//

```

```

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return data[index];
        }//end else
    }//end getData method
    //-----
    ---//

```

```

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            this.data[index] = data;
        }//end else
    }//end setData method
    //-----
    ---//

```

```

    //This method overrides the equals method
    inherited
    // from the class named Object. It compares the
    values
    // stored in two matrices and returns true if the
    // values are equal or almost equal and returns
    false
    // otherwise.
    public boolean equals(Object obj){
        if(obj instanceof GM01.ColMatrix2D &&
            Math.abs(((GM01.ColMatrix2D)obj).getData(0)
-
                                                                getData(0)) <=
0.000001 &&
            Math.abs(((GM01.ColMatrix2D)obj).getData(1)
-
                                                                getData(1)) <=
0.000001){
        return true;
    }

```

```

        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
    ---//

    //Adds one ColMatrix2D object to another
    ColMatrix2D
    // object, returning a ColMatrix2D object.
    public GM01.ColMatrix2D add(GM01.ColMatrix2D
matrix){
        return new GM01.ColMatrix2D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1));
    }//end add
    //-----
    ---//

    //Subtracts one ColMatrix2D object from another
    // ColMatrix2D object, returning a ColMatrix2D
    object.
    // The object that is received as an incoming
    // parameter is subtracted from the object on
    which
    // the method is called.
    public GM01.ColMatrix2D subtract(
                                GM01.ColMatrix2D
matrix){
        return new GM01.ColMatrix2D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1));
    }//end subtract
    //-----

```

```

---//
    }//end class ColMatrix2D

//=====
=//

    //An object of this class represents a 3D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix3D{
        double[] data = new double[3];

        public ColMatrix3D(
                                double data0,double data1,double
data2){
            data[0] = data0;
            data[1] = data1;
            data[2] = data2;
        }//end constructor
        //-----
---//

        public String toString(){
            return data[0] + "," + data[1] + "," + data[2];
        }//end overridden toString method
        //-----
---//

        public double getData(int index){
            if((index < 0) || (index > 2)){
                throw new IndexOutOfBoundsException();
            }else{
                return data[index];
            }//end else
        }//end getData method

```

```

//-----
---//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            this.data[index] = data;
        }//end else
    }//end setData method
//-----
---//

    //This method overrides the equals method
    inherited
    // from the class named Object. It compares the
    values
    // stored in two matrices and returns true if the
    // values are equal or almost equal and returns
    false
    // otherwise.
    public boolean equals(Object obj){
        if(obj instanceof GM01.ColMatrix3D &&
            Math.abs(((GM01.ColMatrix3D)obj).getData(0)
-
                                getData(0)) <=
0.00001 &&
            Math.abs(((GM01.ColMatrix3D)obj).getData(1)
-
                                getData(1)) <=
0.00001 &&
            Math.abs(((GM01.ColMatrix3D)obj).getData(2)
-
                                getData(2)) <=
0.00001){
            return true;
        }else{
            return false;
        }//end else
    }

```

```
    }//end overridden equals method
    //-----
---//
```

```
    //Adds one ColMatrix3D object to another
ColMatrix3D
    // object, returning a ColMatrix3D object.
    public GM01.ColMatrix3D add(GM01.ColMatrix3D
matrix){
    return new GM01.ColMatrix3D(
```

```
getData(0)+matrix.getData(0),
getData(1)+matrix.getData(1),
getData(2)+matrix.getData(2));
    }//end add
    //-----
```

```
---//
```

```
    //Subtracts one ColMatrix3D object from another
    // ColMatrix3D object, returning a ColMatrix3D
object.
```

```
    // The object that is received as an incoming
    // parameter is subtracted from the object on
which
```

```
    // the method is called.
    public GM01.ColMatrix3D subtract(
                                GM01.ColMatrix3D
matrix){
```

```
    return new GM01.ColMatrix3D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1),
                                getData(2)-
matrix.getData(2));
    }//end subtract
```

```

    //-----
---//
    }//end class ColMatrix3D

//=====
=//

//=====
=//

    public static class Point2D{
        GM01.ColMatrix2D point;

        public Point2D(GM01.ColMatrix2D point)
    {//constructor
        //Create and save a clone of the ColMatrix2D
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix2D
object
        // through use of its set method.
        this.point = new ColMatrix2D(
point.getData(0),point.getData(1));
        }//end constructor
    //-----
---//

        public String toString(){
            return point.getData(0) + "," +
point.getData(1);
        }//end toString
    //-----
---//

```



```

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return point.getData(index);
        }//end else
    }//end getData
    //-----
---//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            point.setData(index,data);
        }//end else
    }//end setData
    //-----
---//

    //This method draws a small circle around the
location
    // of the point on the specified graphics
context.
    public void draw(Graphics2D g2D){
        drawOval(g2D,getData(0)-3,
                    getData(1)+3,6,6);
    }//end draw

    //-----
---//

    //Returns a reference to the ColMatrix2D object
that
    // defines this Point2D object.
    public GM01.ColMatrix2D getColMatrix(){
        return point;
    }//end getColMatrix
    //-----

```

```

---//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares the
values
    // stored in the ColMatrix2D objects that define
two
    // Point2D objects and returns true if they are
equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(point.equals(((GM01.Point2D)obj)).

getColMatrix())){
        return true;
    }else{
        return false;
    }//end else

    }//end overridden equals method
    //-----
---//

    //Gets a displacement vector from one Point2D
object
    // to a second Point2D object. The vector points
from
    // the object on which the method is called to
the
    // object passed as a parameter to the method.
Kjell
    // describes this as the distance you would have
to
    // walk along the x and then the y axes to get
from
    // the first point to the second point.
    public GM01.Vector2D getDisplacementVector(
        GM01.Point2D

```

```

point){
    return new GM01.Vector2D(new GM01.ColMatrix2D(
                                point.getData(0)-
getData(0),
                                point.getData(1)-
getData(1)));
    }//end getDisplacementVector
    //-----
    ---//

    //Adds a Vector2D to a Point2D producing a
    // new Point2D.
    public GM01.Point2D addVectorToPoint(
                                GM01.Vector2D
vec){
    return new GM01.Point2D(new GM01.ColMatrix2D(
                                getData(0) +
vec.getData(0),
                                getData(1) +
vec.getData(1)));
    }//end addVectorToPoint
    //-----
    ---//

    //Returns a new Point2D object that is a clone of
    // the object on which the method is called.
    public Point2D clone(){
        return new Point2D(
            new
ColMatrix2D(getData(0),getData(1)));
    }//end clone
    //-----
    ---//

    //The purpose of this method is to rotate a point
    // around a specified anchor point in the x-y
plane.
    //The rotation angle is passed in as a double
value

```

```

// in degrees with the positive angle of rotation
// being counter-clockwise.
//This method does not modify the contents of the
// Point2D object on which the method is called.
// Rather, it uses the contents of that object to
// instantiate, rotate, and return a new Point2D
// object.
//For simplicity, this method translates the
// anchorPoint to the origin, rotates around the
// origin, and then translates back to the
// anchorPoint.
/*

```

See

<http://www.ia.hiof.no/~borres/cgraph/math/threed/p-threed.html> for a definition of the equations required to do the rotation.

```

x2 = x1*cos - y1*sin
y2 = x1*sin + y1*cos
*/

```

```

public GM01.Point2D rotate(GM01.Point2D
anchorPoint,
                        double angle){
    GM01.Point2D newPoint = this.clone();

    double tempX ;
    double tempY;

    //Translate anchorPoint to the origin
    GM01.Vector2D tempVec =
        new
GM01.Vector2D(anchorPoint.getColMatrix());
    newPoint =

newPoint.addVectorToPoint(tempVec.negate());

    //Rotate around the origin.
    tempX = newPoint.getData(0);
    tempY = newPoint.getData(1);

```

```

        newPoint.setData(new x coordinate
                        0,
tempX*Math.cos(angle*Math.PI/180) -
tempY*Math.sin(angle*Math.PI/180));

        newPoint.setData(new y coordinate
                        1,
tempX*Math.sin(angle*Math.PI/180) +
tempY*Math.cos(angle*Math.PI/180));

        //Translate back to anchorPoint
        newPoint = newPoint.addVectorToPoint(tempVec);

        return newPoint;

    } //end rotate
    //-----
---//

    //Multiplies this point by a scaling matrix
    received
    // as an incoming parameter and returns the
    scaled
    // point.
    public GM01.Point2D scale(GM01.ColMatrix2D scale)
    {
        return new GM01.Point2D(new ColMatrix2D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1)));
    } //end scale
    //-----
---//
} //end class Point2D

```

```
//=====
=//
```

```
public static class Point3D{
    GM01.ColMatrix3D point;

    public Point3D(GM01.ColMatrix3D point)
{//constructor
    //Create and save a clone of the ColMatrix3D
object
    // used to define the point to prevent the
point
    // from being corrupted by a later change in
the
    // values stored in the original ColMatrix3D
object
    // through use of its set method.
    this.point =
        new ColMatrix3D(point.getData(0),
                        point.getData(1),
                        point.getData(2));

    }//end constructor
    //-----
---//

    public String toString(){
        return point.getData(0) + "," +
point.getData(1)
                                + "," +
point.getData(2);
    }//end toString
    //-----
---//
```

```
public double getData(int index){
    if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
    }
}
```

```

        }else{
            return point.getData(index);
        }//end else
    }//end getData
    //-----
    ---//

```

```

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            point.setData(index,data);
        }//end else
    }//end setData
    //-----
    ---//

```

//This method draws a small circle around the
 location

// of the point on the specified graphics
 context.

```

    public void draw(Graphics2D g2D){

        //Get 2D projection coordinate values.
        ColMatrix2D temp = convert3Dto2D(point);
        drawOval(g2D,temp.getData(0)-3,
                temp.getData(1)+3,
                6,
                6);

    }//end draw
    //-----
    ---//

```

//Returns a reference to the ColMatrix3D object
 that

// defines this Point3D object.

```

    public GM01.ColMatrix3D getColMatrix(){
        return point;
    }//end getColMatrix

```

```

//-----
---//

//This method overrides the equals method
inherited
// from the class named Object. It compares the
values
// stored in the ColMatrix3D objects that define
two
// Point3D objects and returns true if they are
equal
// and false otherwise.
public boolean equals(Object obj){
    if(point.equals(((GM01.Point3D)obj)).

getColMatrix())){
    return true;
    }else{
    return false;
    }//end else

} //end overridden equals method
//-----
---//

//Gets a displacement vector from one Point3D
object
// to a second Point3D object. The vector points
from
// the object on which the method is called to
the
// object passed as a parameter to the method.
Kjell
// describes this as the distance you would have
to
// walk along the x and then the y axes to get
from
// the first point to the second point.
public GM01.Vector3D getDisplacementVector(

```



```

GM01.Point3D
point){
    return new GM01.Vector3D(new GM01.ColMatrix3D(
        point.getData(0)-
        point.getData(1)-
        point.getData(2))-
        getData(2)));
    }//end getDisplacementVector
    //-----
    ---//

    //Adds a Vector3D to a Point3D producing a
    // new Point3D.
    public GM01.Point3D addVectorToPoint(
        GM01.Vector3D
    vec){
        return new GM01.Point3D(new GM01.ColMatrix3D(
            getData(0) +
            vec.getData(0),
            getData(1) +
            vec.getData(1),
            getData(2) +
            vec.getData(2)));
        }//end addVectorToPoint
        //-----
        ---//

        //Returns a new Point3D object that is a clone of
        // the object on which the method is called.
        public Point3D clone(){
            return new Point3D(new ColMatrix3D(getData(0),
                getData(1),
                getData(2)));
            }//end clone
            //-----
            ---//

```

```

    //The purpose of this method is to rotate a point
    // around a specified anchor point in the
following
    // order:
    // Rotate around z - rotation in x-y plane.
    // Rotate around x - rotation in y-z plane.
    // Rotate around y - rotation in x-z plane.
    //The rotation angles are passed in as double
values
    // in degrees (based on the right-hand rule) in
the
    // order given above, packaged in an object of
the
    // class GM01.ColMatrix3D. (Note that in this
case,
    // the ColMatrix3D object is simply a convenient
    // container and it has no significance from a
matrix
    // viewpoint.)
    //The right-hand rule states that if you point
the
    // thumb of your right hand in the positive
direction
    // of an axis, the direction of positive rotation
    // around that axis is given by the direction
that
    // your fingers will be pointing.
    //This method does not modify the contents of the
    // Point3D object on which the method is called.
    // Rather, it uses the contents of that object to
    // instantiate, rotate, and return a new Point3D
    // object.
    //For simplicity, this method translates the
    // anchorPoint to the origin, rotates around the
    // origin, and then translates back to the
    // anchorPoint.
    /*
See

```

<http://www.ia.hiof.no/~borres/cgraph/math/threed/p-threed.html> for a definition of the equations required to do the rotation.

z-axis

$$x_2 = x_1 \cos v - y_1 \sin v$$

$$y_2 = x_1 \sin v + y_1 \cos v$$

x-axis

$$y_2 = y_1 \cos(v) - z_1 \sin(v)$$

$$z_2 = y_1 \sin(v) + z_1 \cos(v)$$

y-axis

$$x_2 = x_1 \cos(v) + z_1 \sin(v)$$

$$z_2 = -x_1 \sin(v) + z_1 \cos(v)$$

*/

```
public GM01.Point3D rotate(GM01.Point3D
anchorPoint,
                                GM01.ColMatrix3D
angles){
    GM01.Point3D newPoint = this.clone();

    double tempX ;
    double tempY;
    double tempZ;

    //Translate anchorPoint to the origin
    GM01.Vector3D tempVec =
        new
GM01.Vector3D(anchorPoint.getColMatrix());
    newPoint =

newPoint.addVectorToPoint(tempVec.negate());

    double zAngle = angles.getData(0);
    double xAngle = angles.getData(1);
    double yAngle = angles.getData(2);

    //Rotate around z-axis
    tempX = newPoint.getData(0);
```

```

        tempY = newPoint.getData(1);
        newPoint.setData(//new x coordinate
                        0,

tempX*Math.cos(zAngle*Math.PI/180) -
tempY*Math.sin(zAngle*Math.PI/180));

        newPoint.setData(//new y coordinate
                        1,

tempX*Math.sin(zAngle*Math.PI/180) +
tempY*Math.cos(zAngle*Math.PI/180));

        //Rotate around x-axis
        tempY = newPoint.getData(1);
        tempZ = newPoint.getData(2);
        newPoint.setData(//new y coordinate
                        1,

tempY*Math.cos(xAngle*Math.PI/180) -
tempZ*Math.sin(xAngle*Math.PI/180));

        newPoint.setData(//new z coordinate
                        2,

tempY*Math.sin(xAngle*Math.PI/180) +
tempZ*Math.cos(xAngle*Math.PI/180));

        //Rotate around y-axis
        tempX = newPoint.getData(0);
        tempZ = newPoint.getData(2);
        newPoint.setData(//new x coordinate
                        0,

tempX*Math.cos(yAngle*Math.PI/180) +

```

```

tempZ*Math.sin(yAngle*Math.PI/180));

        newPoint.setData(new z coordinate
                        2,
                        -
tempX*Math.sin(yAngle*Math.PI/180) +
tempZ*Math.cos(yAngle*Math.PI/180));

        //Translate back to anchorPoint
        newPoint = newPoint.addVectorToPoint(tempVec);

        return newPoint;

    }//end rotate
    //-----
    ---//

    //Multiplies this point by a scaling matrix
    received
    // as an incoming parameter and returns the
    scaled
    // point.
    public GM01.Point3D scale(GM01.ColMatrix3D scale)
    {
        return new GM01.Point3D(new ColMatrix3D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1),
                                getData(2) *
scale.getData(2)));
    }//end scale
    //-----
    ---//
    }//end class Point3D

//=====

```

```

=//

//=====
=//

    public static class Vector2D{
        GM01.ColMatrix2D vector;

        public Vector2D(GM01.ColMatrix2D vector)
{//constructor
    //Create and save a clone of the ColMatrix2D
object
    // used to define the vector to prevent the
vector
    // from being corrupted by a later change in
the
    // values stored in the original ColVector2D
object.
    this.vector = new ColMatrix2D(
vector.getData(0),vector.getData(1));
    }//end constructor
    //-----
---//

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1);
    }//end toString
    //-----
---//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }
}

```

```

        }//end getData
        //-----
    ---//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData
    //-----
    ---//

    //This method draws a vector on the specified
    graphics
    // context, with the tail of the vector located
    at a
    // specified point, and with a small filled
    circle at
    // the head.
    public void draw(Graphics2D g2D,GM01.Point2D
    tail){

        drawLine(g2D,
            tail.getData(0),
            tail.getData(1),
            tail.getData(0)+vector.getData(0),
            tail.getData(1)+vector.getData(1));

        fillOval(g2D,
            tail.getData(0)+vector.getData(0)-3,
            tail.getData(1)+vector.getData(1)+3,
            6,
            6);
    }//end draw
    //-----
    ---//

```

```

        //Returns a reference to the ColMatrix2D object
that
        // defines this Vector2D object.
        public GM01.ColMatrix2D getColMatrix(){
            return vector;
        }//end getColMatrix
        //-----
---//

        //This method overrides the equals method
inherited
        // from the class named Object. It compares the
values
        // stored in the ColMatrix2D objects that define
two
        // Vector2D objects and returns true if they are
equal
        // and false otherwise.
        public boolean equals(Object obj){
            if(vector.equals((
(GM01.Vector2D)obj).getColMatrix())){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
---//

        //Adds this vector to a vector received as an
incoming
        // parameter and returns the sum as a vector.
        public GM01.Vector2D add(GM01.Vector2D vec){
            return new GM01.Vector2D(new ColMatrix2D(
vec.getData(0)+vector.getData(0),

```



```

vec.getData(1)+vector.getData(1)));
    }//end add
    //-----
---//

    //Returns the length of a Vector2D object.
    public double getLength(){
        return Math.sqrt(
            getData(0)*getData(0) +
            getData(1)*getData(1));
    }//end getLength
    //-----
---//

    //Multiplies this vector by a scale factor
    received as
    // an incoming parameter and returns the scaled
    // vector.
    public GM01.Vector2D scale(Double factor){
        return new GM01.Vector2D(new ColMatrix2D(
            getData(0) *
factor,
            getData(1) *
factor));
    }//end scale
    //-----
---//

    //Changes the sign on each of the vector
    components
    // and returns the negated vector.
    public GM01.Vector2D negate(){
        return new GM01.Vector2D(new ColMatrix2D(
            -
            -
            getData(0),
            -
            -
            getData(1)));
    }//end negate
    //-----

```

```

---//

    //Returns a new vector that points in the same
    // direction but has a length of one unit.
    public GM01.Vector2D normalize(){
        double length = getLength();
        return new GM01.Vector2D(new ColMatrix2D(

getData(0)/length,

getData(1)/length));
    }//end normalize
    //-----
---//
    }//end class Vector2D

//=====
=//

    public static class Vector3D{
        GM01.ColMatrix3D vector;

        public Vector3D(GM01.ColMatrix3D vector)
    {//constructor
        //Create and save a clone of the ColMatrix3D
object
        // used to define the vector to prevent the
vector
        // from being corrupted by a later change in
the
        // values stored in the original ColMatris3D
object.
        this.vector = new
ColMatrix3D(vector.getData(0),
vector.getData(1),
vector.getData(2));

```

```

        }//end constructor
        //-----
    ---//

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1)
                                + "," +
vector.getData(2);
    }//end toString
    //-----
    ---//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
    //-----
    ---//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData
    //-----
    ---//

    //This method draws a vector on the specified
graphics
    // context, with the tail of the vector located
at a
    // specified point, and with a small circle at
the

```

```

        // head.
        public void draw(Graphics2D g2D, GM01.Point3D
tail){

            //Get a 2D projection of the tail
            GM01.ColMatrix2D tail2D =
convert3Dto2D(tail.point);

            //Get the 3D location of the head
            GM01.ColMatrix3D head =

tail.point.add(this.getColMatrix());

            //Get a 2D projection of the head
            GM01.ColMatrix2D head2D = convert3Dto2D(head);
            drawLine(g2D, tail2D.getData(0),
                    tail2D.getData(1),
                    head2D.getData(0),
                    head2D.getData(1));

            //Draw a small filled circle to identify the
head.
            fillOval(g2D, head2D.getData(0)-3,
                    head2D.getData(1)+3,
                    6,
                    6);

        } //end draw
        //-----
---//

        //Returns a reference to the ColMatrix3D object
that
        // defines this Vector3D object.
        public GM01.ColMatrix3D getColMatrix(){
            return vector;
        } //end getColMatrix
        //-----
---//

```

```

        //This method overrides the equals method
inherited
        // from the class named Object. It compares the
values
        // stored in the ColMatrix3D objects that define
two
        // Vector3D objects and returns true if they are
equal
        // and false otherwise.
        public boolean equals(Object obj){
            if(vector.equals((
(GM01.Vector3D)obj).getColMatrix())){
                return true;
            }else{
                return false;
            }//end else

```

```

        }//end overridden equals method
        //-----
---//

```

```

        //Adds this vector to a vector received as an
incoming

```

```

        // parameter and returns the sum as a vector.
        public GM01.Vector3D add(GM01.Vector3D vec){
            return new GM01.Vector3D(new ColMatrix3D(

```

```

vec.getData(0)+vector.getData(0),

```

```

vec.getData(1)+vector.getData(1),

```

```

vec.getData(2)+vector.getData(2)));

```

```

        }//end add

```

```

        //-----
---//

```

```

        //Returns the length of a Vector3D object.

```

```

    public double getLength(){
        return Math.sqrt(getData(0)*getData(0) +
                           getData(1)*getData(1) +
                           getData(2)*getData(2));
    }//end getLength
    //-----
    ---//

    //Multiplies this vector by a scale factor
    received as
    // an incoming parameter and returns the scaled
    // vector.
    public GM01.Vector3D scale(Double factor){
        return new GM01.Vector3D(new ColMatrix3D(
            factor,
            factor,
            factor));
    }//end scale
    //-----
    ---//

    //Changes the sign on each of the vector
    components
    // and returns the negated vector.
    public GM01.Vector3D negate(){
        return new GM01.Vector3D(new ColMatrix3D(
            -
            -
            -
            getData(0),
            getData(1),
            getData(2)));
    }//end negate
    //-----
    ---//

```

```

        //Returns a new vector that points in the same
        // direction but has a length of one unit.
        public GM01.Vector3D normalize(){
            double length = getLength();
            return new GM01.Vector3D(new ColMatrix3D(

getData(0)/length,

getData(1)/length,

getData(2)/length));
        }//end normalize
        //-----
    ---//
    }//end class Vector3D

//=====
=//

//=====
=//

    //A line is defined by two points. One is called
    the
    // tail and the other is called the head. Note that
    this
    // class has the same name as one of the classes in
    // the Graphics2D class. Therefore, if the class
    from
    // the Graphics2D class is used in some future
    upgrade
    // to this program, it will have to be fully
    qualified.
    public static class Line2D{
        GM01.Point2D[] line = new GM01.Point2D[2];

        public Line2D(GM01.Point2D tail,GM01.Point2D
head){

```

```

        //Create and save clones of the points used to
        // define the line to prevent the line from
being
        // corrupted by a later change in the
coordinate
        // values of the points.
        this.line[0] = new Point2D(new
GM01.ColMatrix2D(

tail.getData(0),tail.getData(1)));
        this.line[1] = new Point2D(new
GM01.ColMatrix2D(

head.getData(0),head.getData(1)));
    }//end constructor
    //-----
---//

    public String toString(){
        return "Tail = " + line[0].getData(0) + "," +
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + "," +
            + line[1].getData(1);
    }//end toString
    //-----
---//

    public GM01.Point2D getTail(){
        return line[0];
    }//end getTail
    //-----
---//

    public GM01.Point2D getHead(){
        return line[1];
    }//end getHead
    //-----
---//

```



```

        public void setTail(GM01.Point2D newPoint){
            //Create and save a clone of the new point to
            // prevent the line from being corrupted by a
            // later change in the coordinate values of the
            // point.
            this.line[0] = new Point2D(new
GM01.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
        }//end setTail
        //-----
    ---//

        public void setHead(GM01.Point2D newPoint){
            //Create and save a clone of the new point to
            // prevent the line from being corrupted by a
            // later change in the coordinate values of the
            // point.
            this.line[1] = new Point2D(new
GM01.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
        }//end setHead
        //-----
    ---//

        public void draw(Graphics2D g2D){
            drawLine(g2D,getTail().getData(0),
                    getTail().getData(1),
                    getHead().getData(0),
                    getHead().getData(1));
        }//end draw
        //-----
    ---//
    }//end class Line2D

//=====
=//

```

```

    //A line is defined by two points. One is called
the
    // tail and the other is called the head.
    public static class Line3D{
        GM01.Point3D[] line = new GM01.Point3D[2];

        public Line3D(GM01.Point3D tail,GM01.Point3D
head){
            //Create and save clones of the points used to
            // define the line to prevent the line from
being
            // corrupted by a later change in the
coordinate
            // values of the points.
            this.line[0] = new Point3D(new
GM01.ColMatrix3D(
tail.getData(0),
tail.getData(1),
tail.getData(2)));
            this.line[1] = new Point3D(new
GM01.ColMatrix3D(
head.getData(0),
head.getData(1),
head.getData(2)));
        }//end constructor
        //-----
        ---//

        public String toString(){
            return "Tail = " + line[0].getData(0) + ","
                + line[0].getData(1) + ","
                + line[0].getData(2)

```

```

        + "\nHead = "
        + line[1].getData(0) + ", "
        + line[1].getData(1) + ", "
        + line[1].getData(2);
    }//end toString
    //-----
    ---//

    public GM01.Point3D getTail(){
        return line[0];
    }//end getTail
    //-----
    ---//

    public GM01.Point3D getHead(){
        return line[1];
    }//end getHead
    //-----
    ---//

    public void setTail(GM01.Point3D newPoint){
        //Create and save a clone of the new point to
        // prevent the line from being corrupted by a
        // later change in the coordinate values of the
        // point.
        this.line[0] = new Point3D(new
GM01.ColMatrix3D(
newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
    }//end setTail
    //-----
    ---//

    public void setHead(GM01.Point3D newPoint){
        //Create and save a clone of the new point to

```

```

        // prevent the line from being corrupted by a
        // later change in the coordinate values of the
        // point.
        this.line[1] = new Point3D(new
GM01.ColMatrix3D(

newPoint.getData(0),

newPoint.getData(1),

newPoint.getData(2)));
    }//end setHead
    //-----
---//

    public void draw(Graphics2D g2D){

        //Get 2D projection coordinates.
        GM01.ColMatrix2D tail =

convert3Dto2D(getTail().point);
        GM01.ColMatrix2D head =

convert3Dto2D(getHead().point);

        drawLine(g2D,tail.getData(0),
                    tail.getData(1),
                    head.getData(0),
                    head.getData(1));
    }//end draw
    //-----
---//
} //end class Line3D

//=====
=//

} //end class GM01

```

```
//=====
===//
```



Listing 27 . Source code for the program named GM01test02.

```
/*GM01test02.java
Copyright 2008, R.G.Baldwin
Revised 02/18/08
```

This program tests many 2D aspects of the GM01
library
using both text and graphics.

Tested using JDK 1.6 under WinXP.

```
*****
****/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

class GM01test02{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class GM01test02
//=====
===//
```

```
class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
```

```

GUI(){//constructor
    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create a new drawing canvas and add it to the
    // center of the JFrame.
    myCanvas = new MyCanvas();
    this.getContentPane().add(myCanvas);

    //This object must be visible before you can get
an
    // off-screen image. It must also be visible
before
    // you can compute the size of the canvas.
    setVisible(true);
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

    //Create an off-screen image and get a graphics
    // context on it.
    osi = createImage(osiWidth,osiHeight);
    Graphics2D g2D = (Graphics2D)(osi.getGraphics());

    //Perform tests using text.
    testUsingText();

    //Perform tests using graphics.
    drawOffScreen(g2D);

    //Cause the overridden paint method belonging to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
//-----
---//

```

```

//The purpose of this method is to test various
// 2D aspects of the library using text.
public void testUsingText(){
    System.out.println(
        "Test overridden toString of
ColMatrix2D");
    GM01.ColMatrix2D tempA =
        new
GM01.ColMatrix2D(1.5,2.5);
    System.out.println(tempA);
    System.out.println();

    System.out.println(
        "Test setData and getData of
ColMatrix2D");
    tempA.setData(0,4.5);
    System.out.println(tempA.getData(0));
    tempA.setData(1,5.5);
    System.out.println(tempA.getData(1));
    System.out.println();

    System.out.println(
        "Test equals method of
ColMatrix2D");
    GM01.ColMatrix2D tempB =
        new
GM01.ColMatrix2D(1.5,2.5);
    System.out.println(tempA.equals(tempB));
    tempB.setData(0,4.5);
    tempB.setData(1,5.5);
    System.out.println(tempA.equals(tempB));
    System.out.println();

    System.out.println("Test add method of
ColMatrix2D");
    System.out.println(tempA.add(tempB));
    System.out.println();

```

```

        System.out.println(
            "Test subtract method of
ColMatrix2D");
        System.out.println(tempA.subtract(tempB));
        System.out.println();

        System.out.println("Test toString method of
Point2D");
        GM01.Point2D pointA = new GM01.Point2D(tempA);
        System.out.println(pointA);
        System.out.println();

        System.out.println(
            "Test setData and getData of
Point2D");
        pointA.setData(0,1.1);
        System.out.println(pointA.getData(0));
        pointA.setData(1,2.2);
        System.out.println(pointA.getData(1));
        System.out.println();

        System.out.println(
            "Test getColMatrix method of
Point2D");
        System.out.println(pointA.getColMatrix());
        System.out.println();

        System.out.println("Test equals method of
Point2D");
        GM01.Point2D pointB = new GM01.Point2D(tempB);
        System.out.println(pointA.equals(pointB));
        pointA = new GM01.Point2D(tempB);
        System.out.println(pointA.equals(pointB));
        System.out.println();

        System.out.println(
            "Test getDisplacementVector method of
Point2D");
        pointA =

```



```

        new GM01.Point2D(new
GM01.ColMatrix2D(1.5,2.5));
        System.out.println(pointA);
        System.out.println(pointB);
        System.out.println(

pointA.getDisplacementVector(pointB));
        System.out.println();

        System.out.println(
            "Test addVectorToPoint method of
Point2D");
        System.out.println(pointA);
        System.out.println(pointA.addVectorToPoint(
            new GM01.Vector2D(new
GM01.ColMatrix2D(5.5,6.5))));
        System.out.println();

        //See the method named drawOffScreen for a test
of
        // the draw method of the Point2D class.

        System.out.println(
            "Test toString method of
Vector2D");
        GM01.Vector2D vecA =
            new GM01.Vector2D(new
GM01.ColMatrix2D(1.5,2.5));
        System.out.println(vecA);
        System.out.println();

        System.out.println(
            "Test setData and getData methods of
Vector2D");
        vecA.setData(0,4.5);
        System.out.println(vecA.getData(0));
        vecA.setData(1,5.5);
        System.out.println(vecA.getData(1));
        System.out.println();

```

```

    //See the method named drawOffScreen for a test
of
    // the draw method of the Vector2D class.

    System.out.println(
        "Test getColMatrix method of
Vector2D");
    System.out.println(vecA.getColMatrix());
    System.out.println();

    System.out.println("Test equals method of
Vector2D");
    GM01.Vector2D vecB =
        new GM01.Vector2D(new
GM01.ColMatrix2D(1.5,2.5));
    System.out.println(vecA.equals(vecB));
    vecB.setData(0,4.5);
    vecB.setData(1,5.5);
    System.out.println(vecA.equals(vecB));

    System.out.println("Test add method of
Vector2D");
    System.out.println(vecA);
    vecB =
        new GM01.Vector2D(new
GM01.ColMatrix2D(-1.5,3.5));
    System.out.println(vecB);
    System.out.println(vecA.add(vecB));
    System.out.println();

    System.out.println(
        "Test getLength method of
Vector2D");
    vecA =
        new GM01.Vector2D(new
GM01.ColMatrix2D(3.0,4.0));
    System.out.println(vecA);

```

```

        System.out.println(vecA.getLength());
        System.out.println();

        System.out.println("Test toString method of
Line2D");
        GM01.Line2D lineA = new
GM01.Line2D(pointA,pointB);
        System.out.println(lineA);
        System.out.println();

        System.out.println("Test setTail, setHead,
getTail, "
                           + "\nand getHead methods of
Line2D");
        lineA.setTail(pointB);
        lineA.setHead(pointA);
        System.out.println(lineA.getTail());
        System.out.println(lineA.getHead());
        System.out.println();

        //See the method named drawOffScreen for a test
of
        // the draw method of the Line2D class.

        }//end testUsingText
        //-----
        ---//

        //The purpose of this method is test various 2D
aspects
        // of the game-math library using graphics.
        void drawOffScreen(Graphics2D g2D){

            //Translate the origin on the off-screen
            // image and draw a pair of orthogonal axes on
it.
            setCoordinateFrame(g2D);

            //Define the corners of a square in 2D that is

```

```

        // centered on the origin.
        GM01.Point2D pointA =
            new GM01.Point2D(new
GM01.ColMatrix2D(75, -75));
        GM01.Point2D pointB =
            new GM01.Point2D(new
GM01.ColMatrix2D(-75, -75));
        GM01.Point2D pointC =
            new GM01.Point2D(new
GM01.ColMatrix2D(-75, 75));
        GM01.Point2D pointD =
            new GM01.Point2D(new
GM01.ColMatrix2D(75, 75));

        //Draw three of the points in BLACK. Draw the
fourth
        // point in RED to identify it.
        g2D.setColor(Color.BLACK);
        pointA.draw(g2D);
        pointB.draw(g2D);
        pointC.draw(g2D);
        g2D.setColor(Color.RED);
        pointD.draw(g2D);
        g2D.setColor(Color.BLACK);

        //Draw four lines connecting the points to
produce
        // the outline of a square.
        GM01.Line2D lineAB = new
GM01.Line2D(pointA, pointB);
        lineAB.draw(g2D);

        GM01.Line2D lineBC = new
GM01.Line2D(pointB, pointC);
        lineBC.draw(g2D);

        GM01.Line2D lineCD = new
GM01.Line2D(pointC, pointD);

```

```

        lineCD.draw(g2D);

        GM01.Line2D lineDA = new
GM01.Line2D(pointD,pointA);
        lineDA.draw(g2D);

        //Define a vector.
        GM01.Vector2D vecA = new GM01.Vector2D(
                                new
GM01.ColMatrix2D(75,75));

        //Draw the vector with its tail at pointB. The
length
        // and direction of the vector will cause its
head
        // to be at the origin.
        g2D.setColor(Color.MAGENTA);
        vecA.draw(g2D,pointB);

    }//end drawOffScreen
    //-----
    ---//

    //This method is used to set the origin of the
    // off-screen image and to draw orthogonal 2D axes
on
    // the image that intersect at the origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Translate the origin to the center.
        GM01.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

        //Draw x-axis in RED
        g2D.setColor(Color.RED);
        GM01.drawLine(g2D, -75,0,75,0);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        GM01.drawLine(g2D,0,75,0,-75);
    }

```

```

    }//end setCoordinateFrame method

//=====
=//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method will
be
        // called when the JFrame and the Canvas appear
on the
        // screen or when the repaint method is called on
the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI
//=====
===//

```

Listing 28 . Source code for the program named GM01test01.

```

/*GM01test01.java
Copyright 2008, R.G.Baldwin
Revised 02/18/08

```

This program tests many 3D aspects of the GM01 library using both text and graphics.

Tested using JDK 1.6 under WinXP.

```

*****
****/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

class GM01test01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class GM01test01
//=====
===//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        //This object must be visible before you can get
an
        // off-screen image. It must also be visible

```

```

before
    // you can compute the size of the canvas.
    setVisible(true);
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

    //Create an off-screen image and get a graphics
    // context on it.
    osi = createImage(osiWidth,osiHeight);
    Graphics2D g2D = (Graphics2D)(osi.getGraphics());

    //Test many 3D library features using text.
    testUsingText();

    //Test many 3D library features using graphics.
    drawOffScreen(g2D);

    //Cause the overridden paint method belonging to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
//-----
---//

    //The purpose of this method is to test various
    // 3D aspects of the library using text.
    public void testUsingText(){
        System.out.println(
            "Test overridden toString of
ColMatrix3D");
        GM01.ColMatrix3D tempA =
            new
GM01.ColMatrix3D(1.5,2.5,3.5);
        System.out.println(tempA);
        System.out.println();

        System.out.println(
            "Test setData and getData of

```



```

ColMatrix3D");
    tempA.setData(0,4.5);
    System.out.println(tempA.getData(0));
    tempA.setData(1,5.5);
    System.out.println(tempA.getData(1));
    tempA.setData(2,6.5);
    System.out.println(tempA.getData(2));
    System.out.println();

    System.out.println(
        "Test equals method of
ColMatrix3D");
    GM01.ColMatrix3D tempB =
        new
GM01.ColMatrix3D(1.5,2.5,3.5);
    System.out.println(tempA.equals(tempB));
    tempB.setData(0,4.5);
    tempB.setData(1,5.5);
    tempB.setData(2,6.5);
    System.out.println(tempA.equals(tempB));
    System.out.println();

    System.out.println("Test add method of
ColMatrix3D");
    System.out.println(tempA.add(tempB));
    System.out.println();

    System.out.println(
        "Test subtract method of
ColMatrix3D");
    System.out.println(tempA.subtract(tempB));
    System.out.println();

    System.out.println("Test toString method of
Point3D");
    GM01.Point3D pointA = new GM01.Point3D(tempA);
    System.out.println(pointA);
    System.out.println();

```

```

        System.out.println(
            "Test setData and getData of
Point3D");
        pointA.setData(0,1.1);
        System.out.println(pointA.getData(0));
        pointA.setData(1,2.2);
        System.out.println(pointA.getData(1));
        pointA.setData(2,3.3);
        System.out.println(pointA.getData(2));
        System.out.println();

        System.out.println("Test getColMatrix of
Point3D");
        System.out.println(pointA.getColMatrix());
        System.out.println();

        System.out.println("Test equals method of
Point3D");
        GM01.Point3D pointB = new GM01.Point3D(tempB);
        System.out.println(pointA.equals(pointB));
        pointA = new GM01.Point3D(tempB);
        System.out.println(pointA.equals(pointB));
        System.out.println();

        //See the method named drawOffScreen for a test
of
        // the draw method of the Point3D class.

        System.out.println(
            "Test getDisplacementVector method of
Point3D");
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(1.5,2.5,3.5));
        System.out.println(pointA);
        System.out.println(pointB);
        System.out.println(
pointA.getDisplacementVector(pointB));

```

```

        System.out.println();

        System.out.println(
            "Test addVectorToPoint method of
Point3D");
        System.out.println(pointA);
        System.out.println(pointA.addVectorToPoint(
            new GM01.Vector3D(
                new
GM01.ColMatrix3D(5.5,6.5,7.5))));
        System.out.println();

        System.out.println(
            "Test toString method of
Vector3D");
        GM01.Vector3D vecA = new GM01.Vector3D(
            new
GM01.ColMatrix3D(1.5,2.5,3.5));
        System.out.println(vecA);
        System.out.println();

        System.out.println(
            "Test setData and getData methods of
Vector3D");
        vecA.setData(0,4.5);
        System.out.println(vecA.getData(0));
        vecA.setData(1,5.5);
        System.out.println(vecA.getData(1));
        vecA.setData(2,6.5);
        System.out.println(vecA.getData(2));
        System.out.println();

        //See the method named drawOffScreen for a test
of
        // the draw method of the Vector3D class.

        System.out.println(
            "Test getColMatrix method of
Vector3D");

```

```
System.out.println(vecA.getColMatrix());  
System.out.println();
```

```
System.out.println("Test equals method of  
Vector3D");
```

```
GM01.Vector3D vecB = new GM01.Vector3D(  
    new  
GM01.ColMatrix3D(1.5,2.5,3.5));  
System.out.println(vecA.equals(vecB));  
vecB.setData(0,4.5);  
vecB.setData(1,5.5);  
vecB.setData(2,6.5);  
System.out.println(vecA.equals(vecB));
```

```
System.out.println("Test add method of  
Vector3D");
```

```
System.out.println(vecA);  
vecB = new GM01.Vector3D(  
    new  
GM01.ColMatrix3D(-1.5,2.5,3.5));  
System.out.println(vecB);  
System.out.println(vecA.add(vecB));  
System.out.println();
```

```
System.out.println(  
    "Test getLength method of  
Vector3D");
```

```
vecA = new GM01.Vector3D(  
    new  
GM01.ColMatrix3D(3.0,4.0,5.0));  
System.out.println(vecA);  
System.out.println(vecA.getLength());  
System.out.println();
```

```
System.out.println("Test toString method of  
Line3D");
```

```
GM01.Line3D lineA = new  
GM01.Line3D(pointA,pointB);  
System.out.println(lineA);
```

```

        System.out.println();

        System.out.println("Test setTail, setHead,
getTail, "
                           + "\nand getHead methods of
Line3D");
        lineA.setTail(pointB);
        lineA.setHead(pointA);
        System.out.println(lineA.getTail());
        System.out.println(lineA.getHead());
        System.out.println();
    }//end testUsingText
    //-----
    ---//

    //The purpose of this method is to test various
    // 3D aspects of the library using graphics.
    void drawOffScreen(Graphics2D g2D){

        //Translate the origin on the off-screen
        // image and draw a pair of orthogonal axes on
it.
        setCoordinateFrame(g2D);

        //Define eight points that define the corners of
        // a box in 3D that is centered on the origin.

        GM01.Point3D[] points = new GM01.Point3D[8];
        //Right side
        points[0] =
            new GM01.Point3D(new
GM01.ColMatrix3D(75,75,75));
        points[1] =
            new GM01.Point3D(new
GM01.ColMatrix3D(75,75,-75));
        points[2] =
            new GM01.Point3D(new
GM01.ColMatrix3D(75,-75,-75));
        points[3] =

```

```

        new GM01.Point3D(new
GM01.ColMatrix3D(75, -75, 75));
        //Left side
        points[4] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, 75, 75));
        points[5] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, 75, -75));
        points[6] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, -75, -75));
        points[7] =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75, -75, 75));

        //Draw seven of the points in BLACK
        g2D.setColor(Color.BLACK);
        for(int cnt = 1; cnt < points.length; cnt++){
            points[cnt].draw(g2D);
        } //end for loop

        //Draw the right top front point in RED to
identify
        // it.
        g2D.setColor(Color.RED);
        points[0].draw(g2D);
        g2D.setColor(Color.BLACK);

        //Draw lines that connect the points to define
the
        // twelve edges of the box.
        //Right side
        new GM01.Line3D(points[0], points[1]).draw(g2D);
        new GM01.Line3D(points[1], points[2]).draw(g2D);
        new GM01.Line3D(points[2], points[3]).draw(g2D);
        new GM01.Line3D(points[3], points[0]).draw(g2D);

        //Left side

```

```

new GM01.Line3D(points[4],points[5]).draw(g2D);
new GM01.Line3D(points[5],points[6]).draw(g2D);
new GM01.Line3D(points[6],points[7]).draw(g2D);
new GM01.Line3D(points[7],points[4]).draw(g2D);

//Front
new GM01.Line3D(points[0],points[4]).draw(g2D);
new GM01.Line3D(points[3],points[7]).draw(g2D);

//Back
new GM01.Line3D(points[1],points[5]).draw(g2D);
new GM01.Line3D(points[2],points[6]).draw(g2D);

//Define a vector.
GM01.Vector3D vecA = new GM01.Vector3D(
                                new
GM01.ColMatrix3D(75, -75, -75));

//Draw the vector with its tail at the upper-left
// corner of the box. The length and direction of
the
// vector will cause its head to be at the
origin.
g2D.setColor(Color.MAGENTA);
vecA.draw(g2D,points[4]);

} //end drawOffScreen
//-----
---//

//This method is used to set the origin of the
// off-screen image and to draw orthogonal 3D axes
on
// the off-screen image that intersect at the
origin.
// The lengths of the axes are set so as to match
the
// interior dimensions of the box and points are

```

```

drawn
    // where the axes intersect the surfaces of the
    box.
    private void setCoordinateFrame(Graphics2D g2D){

        //Translate the origin to the center.
        GM01.translate(g2D,0.5*osiWidth, -0.5*osiHeight);

        //Draw x-axis in RED
        g2D.setColor(Color.RED);
        GM01.Point3D pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75,0,0));
        GM01.Point3D pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(75,0,0));
        pointA.draw(g2D);
        pointB.draw(g2D);
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, -75,0));
        pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(0,75,0));
        pointA.draw(g2D);
        pointB.draw(g2D);
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw z-axis in BLUE
        g2D.setColor(Color.BLUE);
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(0,0, -75));
        pointB =
            new GM01.Point3D(new

```



```

GM01.ColMatrix3D(0,0,75));
    pointA.draw(g2D);
    pointB.draw(g2D);
    new GM01.Line3D(pointA,pointB).draw(g2D);

} //end setCoordinateFrame method

//=====
=//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method will
be
    // called when the JFrame and the Canvas appear
on the
    // screen or when the repaint method is called on
the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    } //end overridden paint()

} //end inner class MyCanvas

} //end class GUI
//=====
===//

```

Listing 29 . Source code for the program named GM01test05.

```

/*GM01test05.java
Copyright 2008, R.G.Baldwin
Revised 02/24/08

```

This program illustrates vector addition in 3D.

Tested using JDK 1.6 under WinXP.

****/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
```

```
class GM01test05{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class GM01test05
//=====
===//
```

```
class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 300;
    int vSize = 200;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        //This object must be visible before you can get
```

an

```
// off-screen image. It must also be visible
before
// you can compute the size of the canvas.
setVisible(true);
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

//Create an off-screen image and get a graphics
// context on it.
osi = createImage(osiWidth,osiHeight);
Graphics2D g2D = (Graphics2D)(osi.getGraphics());

//Draw on the off-screen image.
drawOffScreen(g2D);

//Cause the overridden paint method belonging to
// myCanvas to be executed.
myCanvas.repaint();

} //end constructor
//-----
---//
```

```
//The purpose of this method is to illustrate
vector
// addition in 3D
void drawOffScreen(Graphics2D g2D){

    //Translate the origin on the off-screen image,
draw a
    // pair of orthogonal axes on it that intersect
at the
    // origin, and paint the background white.
    setCoordinateFrame(g2D);

    //Define two vectors that will be added.
    GM01.Vector3D vecA = new GM01.Vector3D(
        new
```

```

GM01.ColMatrix3D(75,75,75));

    GM01.Vector3D vecB = new GM01.Vector3D(
                                new
GM01.ColMatrix3D(-15,10,-50));

    //Create a ref point at the origin for
convenience.
    GM01.Point3D zeroPoint = new GM01.Point3D(
                                new
GM01.ColMatrix3D(0,0,0));

    //Draw vecA in MAGENTA with its tail at the
origin.
    g2D.setColor(Color.MAGENTA);
    vecA.draw(g2D,zeroPoint);

    //Draw vecB in LIGHT_GRAY with its tail at the
head
    // of vecA.
    g2D.setColor(Color.LIGHT_GRAY);
    GM01.Point3D temp =
                                new
GM01.Point3D(vecA.getColMatrix());
    vecB.draw(g2D,temp);

    //Draw vecB in LIGHT_GRAY with its tail at the
origin.
    vecB.draw(g2D,zeroPoint);

    //Draw vecA in MAGENTA with its tail at the head
    // of vecB. This completes a trapezoid.
    g2D.setColor(Color.MAGENTA);
    vecA.draw(g2D,new
GM01.Point3D(vecB.getColMatrix()));

    //Add the two vectors.
    GM01.Vector3D sum = vecA.add(vecB);
    //Draw sum in BLACK with its tail at the origin.

```

```

        g2D.setColor(Color.BLACK);
        sum.draw(g2D,zeroPoint);

    }//end drawOffScreen
    //-----
    ---//

    //This method is used to set the origin of the
    // off-screen image, set the background color to
    WHITE,
    // and draw orthogonal 3D axes on the off-screen
    image
    // that intersect at the origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Translate the origin to the center.
        GM01.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

        //Set background color to WHITE.
        g2D.setColor(Color.WHITE);
        GM01.fillRect(
            g2D,-
            osiWidth/2,osiHeight/2,osiWidth,osiHeight);

        //Draw x-axis in RED
        g2D.setColor(Color.RED);
        GM01.Point3D pointA =
            new GM01.Point3D(new
            GM01.ColMatrix3D(-75,0,0));
        GM01.Point3D pointB =
            new GM01.Point3D(new
            GM01.ColMatrix3D(75,0,0));
        pointA.draw(g2D);
        pointB.draw(g2D);
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA =

```

```

        new GM01.Point3D(new
GM01.ColMatrix3D(0, -75, 0));
        pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, 75, 0));
        pointA.draw(g2D);
        pointB.draw(g2D);
        new GM01.Line3D(pointA, pointB).draw(g2D);

        //Draw z-axis in BLUE
        g2D.setColor(Color.BLUE);
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, 0, -75));
        pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, 0, 75));
        pointA.draw(g2D);
        pointB.draw(g2D);
        new GM01.Line3D(pointA, pointB).draw(g2D);

    } //end setCoordinateFrame method

//=====
=//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method will
be
        // called when the JFrame and the Canvas appear
on the
        // screen or when the repaint method is called on
the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi, 0, 0, this);

```

```

        }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI
//=====
===//

```

Listing 30 . Source code for the program named GM01test06.

```

/*GM01test06.java
Copyright 2008, R.G.Baldwin
Revised 02/24/08

```

This program is an update of the program named GM01test01.
 The purpose is to illustrate scaling a geometric object
 by calling the method named GM01.Point3D.scale for
 each
 point that makes up the object.

Tested using JDK 1.6 under WinXP.

```

*****
****/
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

class GM01test06{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class GM01test06
//=====
===//

class GUI extends JFrame{
    //Specify the horizontal and vertical size of a

```

```

JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas

    GUI(){//constructor
        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create a new drawing canvas and add it to the
        // center of the JFrame.
        myCanvas = new MyCanvas();
        this.getContentPane().add(myCanvas);

        //This object must be visible before you can get
an
        // off-screen image.  It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        Graphics2D g2D = (Graphics2D)(osi.getGraphics());

        //Test many 3D library features using text.
        testUsingText();

        //Test many 3D library features using graphics.
        drawOffScreen(g2D);

```



```

        //Cause the overridden paint method belonging to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    ---//

    //The purpose of this method is to test various
    // 3D aspects of the library using text.
    public void testUsingText(){
        System.out.println(
            "Test overridden toString of
ColMatrix3D");
        GM01.ColMatrix3D tempA =
            new
GM01.ColMatrix3D(1.5,2.5,3.5);
        System.out.println(tempA);
        System.out.println();

        System.out.println(
            "Test setData and getData of
ColMatrix3D");
        tempA.setData(0,4.5);
        System.out.println(tempA.getData(0));
        tempA.setData(1,5.5);
        System.out.println(tempA.getData(1));
        tempA.setData(2,6.5);
        System.out.println(tempA.getData(2));
        System.out.println();

        System.out.println(
            "Test equals method of
ColMatrix3D");
        GM01.ColMatrix3D tempB =
            new
GM01.ColMatrix3D(1.5,2.5,3.5);
        System.out.println(tempA.equals(tempB));
    }

```

```
tempB.setData(0,4.5);
tempB.setData(1,5.5);
tempB.setData(2,6.5);
System.out.println(tempA.equals(tempB));
System.out.println();
```

```
System.out.println("Test add method of
ColMatrix3D");
System.out.println(tempA.add(tempB));
System.out.println();
```

```
System.out.println(
    "Test subtract method of
ColMatrix3D");
System.out.println(tempA.subtract(tempB));
System.out.println();
```

```
System.out.println("Test toString method of
Point3D");
GM01.Point3D pointA = new GM01.Point3D(tempA);
System.out.println(pointA);
System.out.println();
```

```
System.out.println(
    "Test setData and getData of
Point3D");
pointA.setData(0,1.1);
System.out.println(pointA.getData(0));
pointA.setData(1,2.2);
System.out.println(pointA.getData(1));
pointA.setData(2,3.3);
System.out.println(pointA.getData(2));
System.out.println();
```

```
System.out.println("Test getColMatrix of
Point3D");
System.out.println(pointA.getColMatrix());
System.out.println();
```

```

        System.out.println("Test equals method of
Point3D");
        GM01.Point3D pointB = new GM01.Point3D(tempB);
        System.out.println(pointA.equals(pointB));
        pointA = new GM01.Point3D(tempB);
        System.out.println(pointA.equals(pointB));
        System.out.println();

        //See the method named drawOffScreen for a test
of
        // the draw method of the Point3D class.

        System.out.println(
            "Test getDisplacementVector method of
Point3D");
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(1.5,2.5,3.5));
        System.out.println(pointA);
        System.out.println(pointB);
        System.out.println(
pointA.getDisplacementVector(pointB));
        System.out.println();

        System.out.println(
            "Test addVectorToPoint method of
Point3D");
        System.out.println(pointA);
        System.out.println(pointA.addVectorToPoint(
            new GM01.Vector3D(
                new
GM01.ColMatrix3D(5.5,6.5,7.5))));
        System.out.println();

        System.out.println(
            "Test toString method of
Vector3D");
        GM01.Vector3D vecA = new GM01.Vector3D(

```

```

                                new
GM01.ColMatrix3D(1.5,2.5,3.5));
    System.out.println(vecA);
    System.out.println();

    System.out.println(
        "Test setData and getData methods of
Vector3D");
    vecA.setData(0,4.5);
    System.out.println(vecA.getData(0));
    vecA.setData(1,5.5);
    System.out.println(vecA.getData(1));
    vecA.setData(2,6.5);
    System.out.println(vecA.getData(2));
    System.out.println();

    //See the method named drawOffScreen for a test
of
    // the draw method of the Vector3D class.

    System.out.println(
        "Test getColMatrix method of
Vector3D");
    System.out.println(vecA.getColMatrix());
    System.out.println();

    System.out.println("Test equals method of
Vector3D");
    GM01.Vector3D vecB = new GM01.Vector3D(
                                new
GM01.ColMatrix3D(1.5,2.5,3.5));
    System.out.println(vecA.equals(vecB));
    vecB.setData(0,4.5);
    vecB.setData(1,5.5);
    vecB.setData(2,6.5);
    System.out.println(vecA.equals(vecB));

    System.out.println("Test add method of
Vector3D");

```

```

        System.out.println(vecA);
        vecB = new GM01.Vector3D(
                                new
GM01.ColMatrix3D(-1.5,2.5,3.5));
        System.out.println(vecB);
        System.out.println(vecA.add(vecB));
        System.out.println();

        System.out.println(
                                "Test getLength method of
Vector3D");
        vecA = new GM01.Vector3D(
                                new
GM01.ColMatrix3D(3.0,4.0,5.0));
        System.out.println(vecA);
        System.out.println(vecA.getLength());
        System.out.println();

        System.out.println("Test toString method of
Line3D");
        GM01.Line3D lineA = new
GM01.Line3D(pointA,pointB);
        System.out.println(lineA);
        System.out.println();

        System.out.println("Test setTail, setHead,
getTail, "
                                + "\nand getHead methods of
Line3D");
        lineA.setTail(pointB);
        lineA.setHead(pointA);
        System.out.println(lineA.getTail());
        System.out.println(lineA.getHead());
        System.out.println();
    }//end testUsingText
    //-----
    ---//

    //The purpose of this method is to test various

```

```

// 3D aspects of the library using graphics.
void drawOffScreen(Graphics2D g2D){

    //Translate the origin on the off-screen
    // image and draw a pair of orthogonal axes on
it.
    setCoordinateFrame(g2D);

    //Define eight points that define the corners of
    // a box in 3D that is centered on the origin.

    GM01.Point3D[] points = new GM01.Point3D[8];
    //Right side
    points[0] =
        new GM01.Point3D(new
GM01.ColMatrix3D(75,75,75));
    points[1] =
        new GM01.Point3D(new
GM01.ColMatrix3D(75,75,-75));
    points[2] =
        new GM01.Point3D(new
GM01.ColMatrix3D(75,-75,-75));
    points[3] =
        new GM01.Point3D(new
GM01.ColMatrix3D(75,-75,75));
    //Left side
    points[4] =
        new GM01.Point3D(new
GM01.ColMatrix3D(-75,75,75));
    points[5] =
        new GM01.Point3D(new
GM01.ColMatrix3D(-75,75,-75));
    points[6] =
        new GM01.Point3D(new
GM01.ColMatrix3D(-75,-75,-75));
    points[7] =
        new GM01.Point3D(new
GM01.ColMatrix3D(-75,-75,75));

```

```

        //Scale each of the points that define the
corners of
        // the box.
        for(int cnt = 0;cnt < points.length;cnt++){
            points[cnt] = points[cnt].scale(
                new
GM01.ColMatrix3D(0.25,0.5,0.75));
        }//end for loop

        //Draw seven of the points in BLACK
        g2D.setColor(Color.BLACK);
        for(int cnt = 1;cnt < points.length;cnt++){
            points[cnt].draw(g2D);
        }//end for loop

        //Draw the right top front point in RED to
identify
        // it.
        g2D.setColor(Color.RED);
        points[0].draw(g2D);
        g2D.setColor(Color.BLACK);

        //Draw lines that connect the points to define
the
        // twelve edges of the box.
        //Right side
        new GM01.Line3D(points[0],points[1]).draw(g2D);
        new GM01.Line3D(points[1],points[2]).draw(g2D);
        new GM01.Line3D(points[2],points[3]).draw(g2D);
        new GM01.Line3D(points[3],points[0]).draw(g2D);

        //Left side
        new GM01.Line3D(points[4],points[5]).draw(g2D);
        new GM01.Line3D(points[5],points[6]).draw(g2D);
        new GM01.Line3D(points[6],points[7]).draw(g2D);
        new GM01.Line3D(points[7],points[4]).draw(g2D);

        //Front
        new GM01.Line3D(points[0],points[4]).draw(g2D);

```

```

        new GM01.Line3D(points[3],points[7]).draw(g2D);

        //Back
        new GM01.Line3D(points[1],points[5]).draw(g2D);
        new GM01.Line3D(points[2],points[6]).draw(g2D);

    }//end drawOffScreen
    //-----
    ---//

    //This method is used to set the origin of the
    // off-screen image and to draw orthogonal 3D axes
on
    // the off-screen image that intersect at the
origin.
    // Points are drawn where the axes intersect the
    // surfaces of the box.
    private void setCoordinateFrame(Graphics2D g2D){

        //Translate the origin to the center.
        GM01.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

        //Draw x-axis in RED. Scale the points before
        // drawing them so that they will identify the
        // locations where the axes intersect the surface
        // of the box.
        g2D.setColor(Color.RED);
        GM01.Point3D pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(-75,0,0));
        GM01.Point3D pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(75,0,0));
        pointA.scale(
            new
GM01.ColMatrix3D(0.25,0.0,0.0)).draw(g2D);
        pointB.scale(
            new
GM01.ColMatrix3D(0.25,0.0,0.0)).draw(g2D);

```



```

        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, -75, 0));
        pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, 75, 0));
        pointA.scale(
            new
GM01.ColMatrix3D(0.0, 0.5, 0.0)).draw(g2D);
        pointB.scale(
            new
GM01.ColMatrix3D(0.0, 0.5, 0.0)).draw(g2D);
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw z-axis in BLUE
        g2D.setColor(Color.BLUE);
        pointA =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, 0, -75));
        pointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(0, 0, 75));
        pointA.scale(
            new
GM01.ColMatrix3D(0.0, 0.0, 0.75)).draw(g2D);
        pointB.scale(
            new
GM01.ColMatrix3D(0.0, 0.0, 0.75)).draw(g2D);
        new GM01.Line3D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method

    //=====
    ==//

```

```

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method will
be
        // called when the JFrame and the Canvas appear
on the
        // screen or when the repaint method is called on
the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
//=====
===//

```

Listing 31 . Source code for the program named StringArt02.

```

/*StringArt02.java
Copyright 2008, R.G.Baldwin
Revised 02/22/08

```

This is a 2D version of a string art program that supports rotation in two dimensions.

This program produces a 2D string art image by connecting various points that are equally spaced on the circumference of a circle.

Initially, the circle is centered on the origin. There are

six points on the circle connected by lines forming a hexagon, The lines that connect the points are different colors. The radius of the circle is 50 units. The points at the vertices of the hexagon are not drawn, but the lines that connect the vertices are drawn.

A GUI is provided that allows the user to specify the following items and click a Replot button to cause the drawing to change:

- Number Points
- Number Loops
- Rotation angle (deg)
- X anchor point
- Y anchor point

Changing the number of points causes the number of vertices that describe the geometric object to change.

Changing the number of loops causes the number of lines that are drawn to connect the vertices to change. For a value of 1, each vertex is connected to the one next to it. For a value of 2, additional lines are drawn connecting every other vertex. For a value of 3, additional lines are drawn connecting every third vertex, etc.

The image can be rotated around an anchor point. Entering a non-zero value in the Rotation field causes the

image
to be rotated by the specified angle around the
anchor
point.

The anchor point is initially located at the origin,
but
the location of the anchor point can be changed by
the
user. If the anchor point is at the origin, the image
is
rotated around the origin. Otherwise, the image is
rotated
around the point in 2D space specified by the anchor
point. The anchor point is drawn in black.

The rotation angle is specified in degrees with a
positive
angle being counter-clockwise.

The number of points is initially set to six and the
number of loops is initially set to one. Making the
number of points larger and making the number of
loops
larger produces many interesting patterns.

Tested using JDK 1.6 under WinXP.

****/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;
```

```
class StringArt02{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class StringArt02
```

```
//=====
===//
```

```
class GUI extends JFrame implements ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 300;
    int vSize = 470;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.

    JTextField numberPointsField;//User input field.
    JTextField loopsField;//User input field.
    int numberPoints = 6;//Can be modified by the user.
    int loopLim = 1;//Can be modified by the user.

    JTextField rotationField;//User input field
    double rotation;//Rotation angle in degrees
    clockwise.

    JTextField xAnchorPointField;//User input field
    JTextField yAnchorPointField;//User input field
    double xAnchorPoint;//Rotation anchor point.
    double yAnchorPoint;//Rotation anchor point.

    //The following variable is used to refer to an
    array
    // object containing the points that define the
    // vertices of a geometric object.
    GM01.Point2D[] points;

    //-----
    ---//
```

```
    GUI(){//constructor
```

```

//Instantiate the array object that will be used
to
// store the points that define the vertices of
the
// geometric object.
points = new GM01.Point2D[numberPoints];

//Set JFrame size, title, and close operation.
setSize(hSize,vSize);
setTitle("Copyright 2008,R.G.Baldwin");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Instantiate the user input components.
numberPointsField =new JTextField("6");
loopsField = new JTextField("1");
rotationField = new JTextField("0");
xAnchorPointField = new JTextField("0");
yAnchorPointField = new JTextField("0");
JButton button = new JButton("Replot");

//Instantiate a JPanel that will house the user
input
// components and set its layout manager.
JPanel controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(0,2));

//Add the user input components and appropriate
labels
// to the control panel.
controlPanel.add(new JLabel(" Number Points"));
controlPanel.add(numberPointsField);
controlPanel.add(new JLabel(" Number Loops"));
controlPanel.add(loopsField);
controlPanel.add(new JLabel(
                                " Rotation angle
(deg)"));
controlPanel.add(rotationField);
controlPanel.add(new JLabel(" X anchor point"));
controlPanel.add(xAnchorPointField);

```

```

        controlPanel.add(new JLabel(" Y anchor point"));
        controlPanel.add(yAnchorPointField);
        controlPanel.add(button);

        //Add the control panel to the SOUTH position in
the
        // JFrame.
        this.getContentPane().add(

BorderLayout.SOUTH,controlPanel);

        //Create a new drawing canvas and add it to the
        // CENTER of the JFrame above the control panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(

BorderLayout.CENTER,myCanvas);

        //This object must be visible before you can get
an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Erase the off-screen image, establish the
origin,
        // and draw the axes
        setCoordinateFrame(g2D,true);

```

```

        //Create the Point2D objects that define the
geometric
        // object and manipulate them to produce the
desired
        // results.
        drawOffScreen(g2D);

        //Register this object as an action listener on
the
        // button.
        button.addActionListener(this);

        //Cause the overridden paint method belonging to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    ---//

    //The purpose of this method is to Create the
Point2D
    // objects that define the vertices of a geometric
    // object and manipulate them to produce the
desired
    // results.
    void drawOffScreen(Graphics2D g2D){
        //Erase the off-screen image and draw new axes,
but
        // don't move the origin.
        setCoordinateFrame(g2D,false);

        //Create a set of Point2D objects that specify
        // locations on the circumference of a circle
that
        // is in the x-y plane with a radius of 50 units.
Save
        // references to the Point2D objects in an array.

```



```

        for(int cnt = 0;cnt < numberPoints;cnt++){
            points[cnt] = new GM01.Point2D(new
GM01.ColMatrix2D(

50*Math.cos((cnt*360/numberPoints)*Math.PI/180),

50*Math.sin((cnt*360/numberPoints)*Math.PI/180)));

            //The following object is populated with the 2D
            // coordinates of the point around which the
            //rotations will take place.
            GM01.Point2D anchorPoint = new GM01.Point2D(
                new GM01.ColMatrix2D(

xAnchorPoint,yAnchorPoint));
            //Draw the anchorPoint in BLACK.
            g2D.setColor(Color.BLACK);
            anchorPoint.draw(g2D);

            //The following statement causes the rotation
to be
            //performed.
            points[cnt] =

points[cnt].rotate(anchorPoint,rotation);

        }//end for loop

        //Implement the algorithm that draws lines
connecting
        // points on the geometric object.
        GM01.Line2D line;

        //Begin the outer loop.
        for(int loop = 1;loop <= loopLim;loop++){
            //The following variable specifies the array
            // element containing a point on which a line
will
            // start.

```

```

    int start = -1;

    //The following variable specifies the number
of
    // points that will be skipped between the
starting
    // point and the ending point for a line.
    int skip = loop;
    //The following logic causes the element number
to
    // wrap around when it reaches the end of the
    // array.
    while(skip >= 2*numberPoints-1){
        skip -= numberPoints;
    }//end while loop

    //The following variable specifies the array
    // element containing a point on which a line
will
    // end.
    int end = start + skip;

    //Begin inner loop. This loop actually
constructs
    // the GM01.Line2D objects and causes visual
    // manifestations of those objects to be drawn
on
    // the off-screen image. Note the requirement
to
    // wrap around when the element numbers exceed
the
    // length of the array.
    for(int cnt = 0; cnt < numberPoints; cnt++){
        if(start < numberPoints-1){
            start++;
        }else{
            //Wrap around
            start -= (numberPoints-1);
        }//end else

```

```

        if(end < numberPoints-1){
            end++;
        }else{
            //Wrap around.
            end -= (numberPoints-1);
        }//end else

        //Create some interesting colors.
        g2D.setColor(new Color(cnt*255/numberPoints,
127+cnt*64/numberPoints,
                                255-
cnt*255/numberPoints));

        //Create a line that connects points on the
        // geometric object.
        line = new
GM01.Line2D(points[start],points[end]);
        line.draw(g2D);
    }//end inner loop
} //end outer loop

} //end drawOffScreen
//-----
---//

//This method is used to set the origin of the
// off-screen image to the center and to draw
orthogonal
// 2D axes on the image that intersect at the
origin.
//The second parameter is used to determine if the
// origin should be translated to the center.
private void setCoordinateFrame(
                                Graphics2D g2D,boolean
translate){

    //Translate the origin to the center if translate

```

```

is
    // true.
    if(translate){

GM01.translate(g2D,0.5*osiWidth,-0.5*osiHeight);
    }//end if

    //Erase the screen
    g2D.setColor(Color.WHITE);
    GM01.fillRect(g2D,-osiWidth/2,osiHeight/2,

osiWidth,osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM01.Point2D pointA = new GM01.Point2D(
        new GM01.ColMatrix2D(-
osiWidth/2,0));
    GM01.Point2D pointB = new GM01.Point2D(
        new
GM01.ColMatrix2D(osiWidth/2,0));
    new GM01.Line2D(pointA,pointB).draw(g2D);

    //Draw y-axis in GREEN
    g2D.setColor(Color.GREEN);
    pointA = new GM01.Point2D(
        new GM01.ColMatrix2D(0,-
osiHeight/2));
    pointB = new GM01.Point2D(
        new
GM01.ColMatrix2D(0,osiHeight/2));
    new GM01.Line2D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method
    //-----
    ---//

    //This method is called to respond to a click on
the

```

```

    // button.
    public void actionPerformed(ActionEvent e){
        //Get user input values and use them to modify
        several
        // values that control the drawing.
        numberPoints = Integer.parseInt(
numberPointsField.getText());

        loopLim = Integer.parseInt(loopsField.getText());

        //Rotation angle in degrees.
        rotation =
Double.parseDouble(rotationField.getText());

        //Rotation anchor points
        xAnchorPoint =
Double.parseDouble(xAnchorPointField.getText());
        yAnchorPoint =
Double.parseDouble(yAnchorPointField.getText());

        //Instantiate a new array object with a length
        // that matches the new value for numberPoints.
        points = new GM01.Point2D[numberPoints];

        //Draw a new off-screen image based on user
        inputs.
        drawOffScreen(g2D);
        myCanvas.repaint();//Copy off-screen image to
        canvas.
    }//end actionPerformed

//=====
=//

```


```

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method will
be
        // called when the JFrame and the Canvas appear
on the
        // screen or when the repaint method is called on
the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI
//=====
===//

```



Listing 32 . Source code for the program named StringArt03.

```

/*StringArt03.java
Copyright 2008, R.G.Baldwin
Revised 02/22/08

```

This is a 3D version of a string art program that supports rotation in three dimensions.

This program produces a 3D string art image by connecting various points that are equally spaced on the circumference of a circle.

Initially, the circle is on the x-y plane centered on the

origin. There are six points on the circle connected by lines forming a hexagon, The lines that connect the points are different colors. The radius of the circle is 50 units. The points at the vertices of the hexagon are not drawn, but the lines that connect the vertices are drawn.

A GUI is provided that allows the user to specify the following items and click a Replot button to cause the drawing to change:

- Number Points
- Number Loops
- Rotate around Z (deg)
- Rotate around X (deg)
- Rotate around Y (deg)
- X Anchor point
- Y Anchor point
- Z Anchor point

Changing the number of points causes the number of vertices that describe the geometric object to change.

Changing the number of loops causes the number of lines that are drawn to connect the vertices to change. For a value of 1, each vertex is connected to the one next to it. For a value of 2, additional lines are drawn connecting every other vertex. For a value of 3, additional lines are drawn connecting every third vertex, etc.

The image can be rotated in any or all of three dimensions around an anchor point. Entering a non-zero value in one or more of the Rotate fields causes the image to be rotated by the specified angle or angles around the anchor point. The anchor point is initially specified to be at the origin, but the location of the anchor point can be changed by the user. If the anchor point is at the origin, the image is rotated around the origin. Otherwise, the image is rotated around the point in 3D space specified by the anchor point. The anchor point is drawn in black.

The rotation angle is specified in degrees with a positive angle being given by the right-hand rule as applied to the axis around which the image is being rotated.

The rotational effects are cumulative. The image is first rotated around the anchor point in a direction consistent with rotation around the z-axis (rotation in the x-y plane). Then that rotated image is rotated in a direction consistent with rotation around the x-axis (rotation in the y-z plane). Finally, the previously rotated image is rotated in a direction consistent with rotation around the y-axis (rotation in the x-z plane). It is important

to note, however, that the actual rotation is around the anchor point and not around the origin unless the anchor point is at the origin.

The number of points is initially set to six and the number of loops is initially set to one. Making the number of points larger and making the number of loops larger produces many interesting patterns.

Tested using JDK 1.6 under WinXP.

****/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;
```

```
class StringArt03{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class StringArt03
//=====
===//
```

```
class GUI extends JFrame implements ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 300;
    int vSize = 470;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//Off-screen graphics context.
```

```

JTextField numberPointsField;//User input field.
JTextField loopsField;//User input field.
int numberPoints = 6;//Can be modified by the user.
int loopLim = 1;//Can be modified by the user.

JTextField zRotationField;//User input field
JTextField xRotationField;//User input field
JTextField yRotationField;//User input field
double zRotation;//Rotation around z in degrees.
double xRotation;//Rotation around x in degrees.
double yRotation;//Rotation around y in degrees

JTextField xAnchorPointField;//User input field
JTextField yAnchorPointField;//User input field
JTextField zAnchorPointField;//User input field
double xAnchorPoint;//Rotation anchor point.
double yAnchorPoint;//Rotation anchor point.
double zAnchorPoint;//Rotation anchor point.

//The following variable is used to refer to an
array
// object containing the points that define the
// vertices of a geometric object.
GM01.Point3D[] points;

//-----
---//

GUI(){//constructor
    //Instantiate the array object that will be used
to
    // store the points that define the vertices of
the
    // geometric object.
    points = new GM01.Point3D[numberPoints];

    //Set JFrame size, title, and close operation.

```



```

        controlPanel.add(yRotationField);
        controlPanel.add(new JLabel(" X anchor point"));
        controlPanel.add(xAnchorPointField);
        controlPanel.add(new JLabel(" Y anchor point"));
        controlPanel.add(yAnchorPointField);
        controlPanel.add(new JLabel(" Z anchor point"));
        controlPanel.add(zAnchorPointField);
        controlPanel.add(button);

        //Add the control panel to the SOUTH position in
the
        // JFrame.
        this.getContentPane().add(

BorderLayout.SOUTH,controlPanel);

        //Create a new drawing canvas and add it to the
        // CENTER of the JFrame above the control panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(

BorderLayout.CENTER,myCanvas);

        //This object must be visible before you can get
an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

```

```

        //Erase the off-screen image, establish the
origin,
        // and draw the axes
        setCoordinateFrame(g2D,true);

        //Create the Point3D objects that define the
geometric
        // object and manipulate them to produce the
desired
        // results.
        drawOffScreen(g2D);

        //Register this object as an action listener on
the
        // button.
        button.addActionListener(this);

        //Cause the overridden paint method belonging to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    ---//

    //The purpose of this method is to Create the
Point3D
    // objects that define the vertices of a geometric
    // object and manipulate them to produce the
desired
    // results.
    void drawOffScreen(Graphics2D g2D){
        //Erase the off-screen image and draw new axes,
but
        // don't move the origin.
        setCoordinateFrame(g2D,false);

```

```

        //Create a set of Point3D objects that specify
        // locations on the circumference of a circle
that
        // is in the x-y plane with a radius of 50 units.
Save
        // references to the Point3D objects in an array.
        for(int cnt = 0;cnt < numberPoints;cnt++){
            points[cnt] = new GM01.Point3D(
                new GM01.ColMatrix3D(

50*Math.cos((cnt*360/numberPoints)*Math.PI/180),

50*Math.sin((cnt*360/numberPoints)*Math.PI/180),
                0.0));

        //The following ColMatrix3D object must be
populated
        // with three rotation angles in degrees that
        // specify the following rotational angles in
order
        // according to the right-hand rule as applied
to
        // the indicated axis.
        // Rotate around z
        // Rotate around x
        // Rotate around y
        GM01.ColMatrix3D angles = new GM01.ColMatrix3D(

zRotation,xRotation,yRotation);

        //The following object contains the 3D
coordinates
        // of the point around which the rotations will
        // take place.
        GM01.Point3D anchorPoint = new GM01.Point3D(
            new GM01.ColMatrix3D(

xAnchorPoint,yAnchorPoint,zAnchorPoint));
        //Draw the anchorPoint in BLACK.

```

```

        g2D.setColor(Color.BLACK);
        anchorPoint.draw(g2D);

        //The following statement causes the rotation
to be
        //performed.
        points[cnt] =

points[cnt].rotate(anchorPoint,angles);

    }//end for loop

    //Implement the algorithm that draws lines
connecting
    // points on the geometric object.
    GM01.Line3D line;

    //Begin the outer loop.
    for(int loop = 1;loop <= loopLim;loop++){
        //The following variable specifies the array
        // element containing a point on which a line
will
        // start.
        int start = -1;

        //The following variable specifies the number
of
        // points that will be skipped between the
starting
        // point and the ending point for a line.
        int skip = loop;
        //The following logic causes the element number
to
        // wrap around when it reaches the end of the
        // array.
        while(skip >:= 2*numberPoints-1){
            skip -= numberPoints;
        }//end while loop

```

```

//The following variable specifies the array
// element containing a point on which a line
will
// end.
int end = start + skip;

//Begin inner loop. This loop actually
constructs
// the GM01.Line3D objects and causes visual
// manifestations of those objects to be drawn
on
// the off-screen image. Note the requirement
to
// wrap around when the element numbers exceed
the
// length of the array.
for(int cnt = 0;cnt < numberPoints;cnt++){
    if(start < numberPoints-1){
        start++;
    }else{
        //Wrap around
        start -= (numberPoints-1);
    }//end else

    if(end < numberPoints-1){
        end++;
    }else{
        //Wrap around.
        end -= (numberPoints-1);
    }//end else

    //Create some interesting colors.
    g2D.setColor(new Color(cnt*255/numberPoints,
127+cnt*64/numberPoints,
                                255-
cnt*255/numberPoints));

    //Create a line that connects points on the

```



```

        // geometric object.
        line = new
GM01.Line3D(points[start],points[end]);
        line.draw(g2D);
    }//end inner loop
} //end outer loop

} //end drawOffScreen
//-----
---//

```

```

//This method is used to set the origin of the
// off-screen image to the center and to draw
orthogonal
// 3D axes on the image that intersect at the
origin.
//The second parameter is used to determine if the
// origin should be translated to the center.
private void setCoordinateFrame(
                                Graphics2D g2D,boolean
translate){

    //Translate the origin to the center if translate
is
    // true.
    if(translate){
GM01.translate(g2D,0.5*osiWidth,-0.5*osiHeight);
    } //end if

    //Erase the screen
    g2D.setColor(Color.WHITE);
    GM01.fillRect(g2D, -osiWidth/2,osiHeight/2,
osiWidth,osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM01.Point3D pointA = new GM01.Point3D(

```

```

        new GM01.ColMatrix3D(-
osiWidth/2,0,0));
        GM01.Point3D pointB = new GM01.Point3D(
            new
GM01.ColMatrix3D(osiWidth/2,0,0));
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA = new GM01.Point3D(
            new GM01.ColMatrix3D(0,-
osiHeight/2,0));
        pointB = new GM01.Point3D(
            new
GM01.ColMatrix3D(0,osiHeight/2,0));
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw z-axis in BLUE. Make its length the same
as the
        // length of the x-axis.
        g2D.setColor(Color.BLUE);
        pointA = new GM01.Point3D(
            new GM01.ColMatrix3D(0,0,-
osiWidth/2));
        pointB = new GM01.Point3D(
            new
GM01.ColMatrix3D(0,0,osiWidth/2));
        new GM01.Line3D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method
    //-----
    ---//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){
        //Get user input values and use them to modify
several

```

```

        // values that control the drawing.
        numberPoints = Integer.parseInt(
numberPointsField.getText());

        loopLim = Integer.parseInt(loopsField.getText());

        //Rotation around z in degrees.
        zRotation =

Double.parseDouble(zRotationField.getText());
        //Rotation around x in degrees.
        xRotation =

Double.parseDouble(xRotationField.getText());
        //Rotation around y in degrees
        yRotation =

Double.parseDouble(yRotationField.getText());

        //Rotation anchor points
        xAnchorPoint =

Double.parseDouble(xAnchorPointField.getText());
        yAnchorPoint =

Double.parseDouble(yAnchorPointField.getText());
        zAnchorPoint =

Double.parseDouble(zAnchorPointField.getText());

        //Instantiate a new array object with a length
        // that matches the new value for numberPoints.
        points = new GM01.Point3D[numberPoints];

        //Draw a new off-screen image based on user
inputs.
        drawOffScreen(g2D);
        myCanvas.repaint();//Copy off-screen image to

```

```

canvas.
    }//end actionPerformed

//=====
=//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method will
be
        // called when the JFrame and the Canvas appear
on the
        // screen or when the repaint method is called on
the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

}//end class GUI
//=====
===//

```

Exercises

Exercise 1

Using Java and the game-math library named **GM01** , or using a different programming environment of your choice, write a program that creates and draws a 20-level stepped pyramid as shown in [Figure 20](#).

Each level is a box. The dimensions of the bottom box are:

- width = 100 pixels
- height = 200/20 pixels
- depth = 100 pixels

The height of each box is the same. The decrease in the width and depth dimensions of the boxes is linear going from the largest box at the bottom to the smallest box at the top.

The sides of the pyramid may be transparent or opaque - your choice.

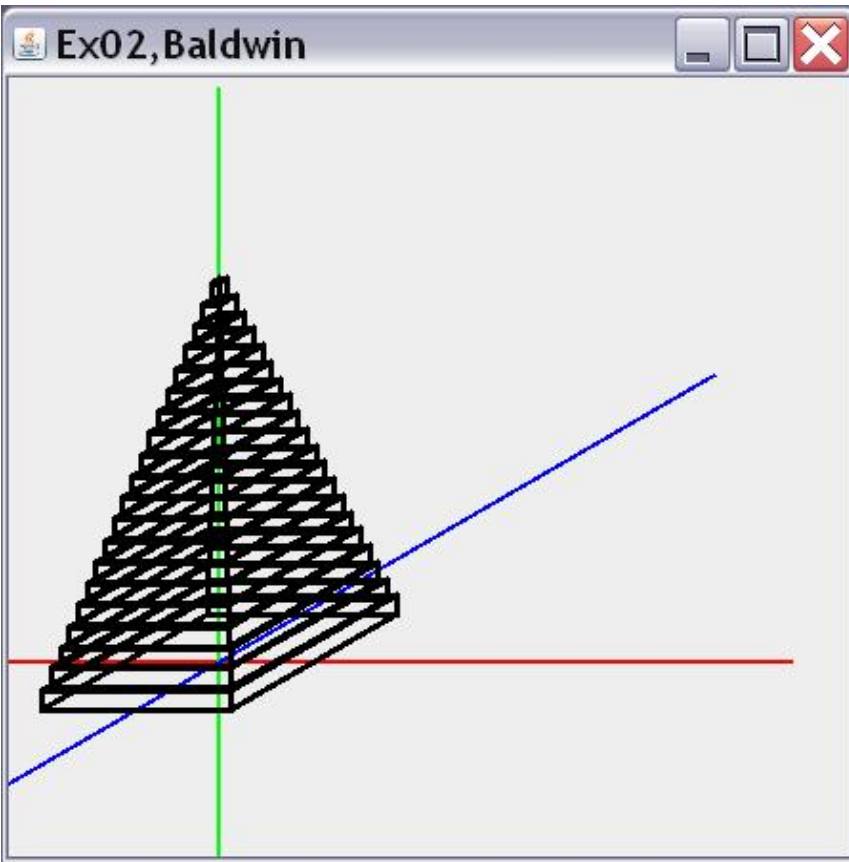
The bottom box sits on the x-z plane, is centered on the vertical axis, and the sides are parallel with the x and z axes.

Draw the axes in the approximate location shown in [Figure 20](#) in red, green, and blue.

The positive direction for x is to the right. The positive direction for y is up, and the positive direction for z protrudes from the screen (down and to the left).

Display your name in the drawing in some manner.

Figure 20 Screen output from Exercise 1.

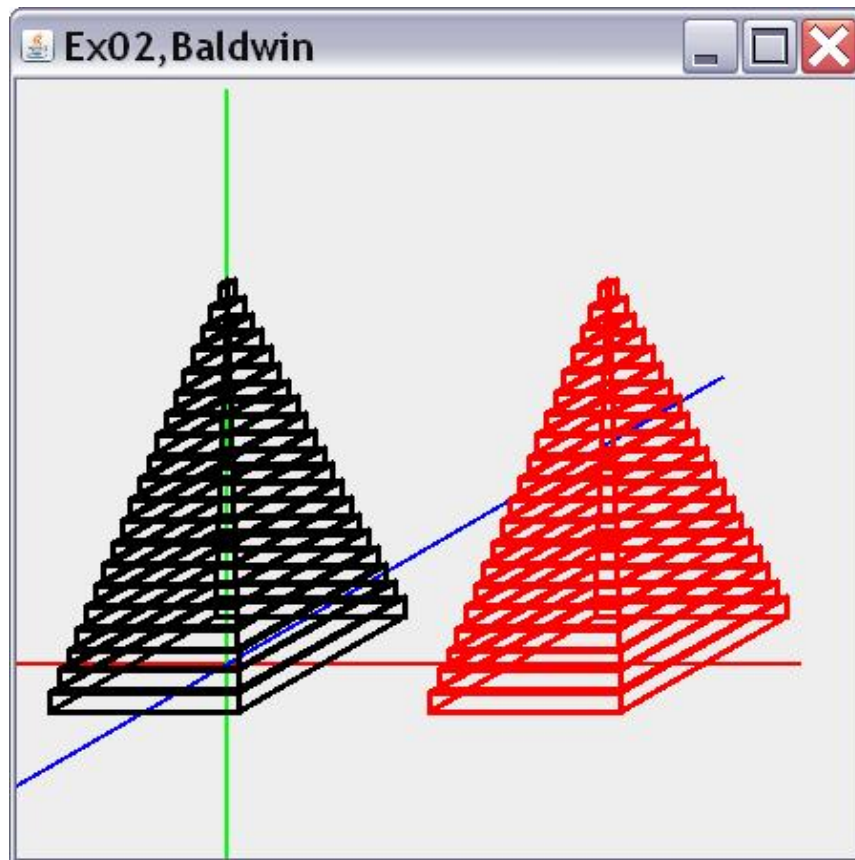


Exercise 2

Beginning with the pyramid that you created in [Exercise 1](#), create a replica of that pyramid positioned at a point that is twice the width of the bottom box from the origin in the positive x direction.

Draw that pyramid in red as shown in [Figure 21](#).

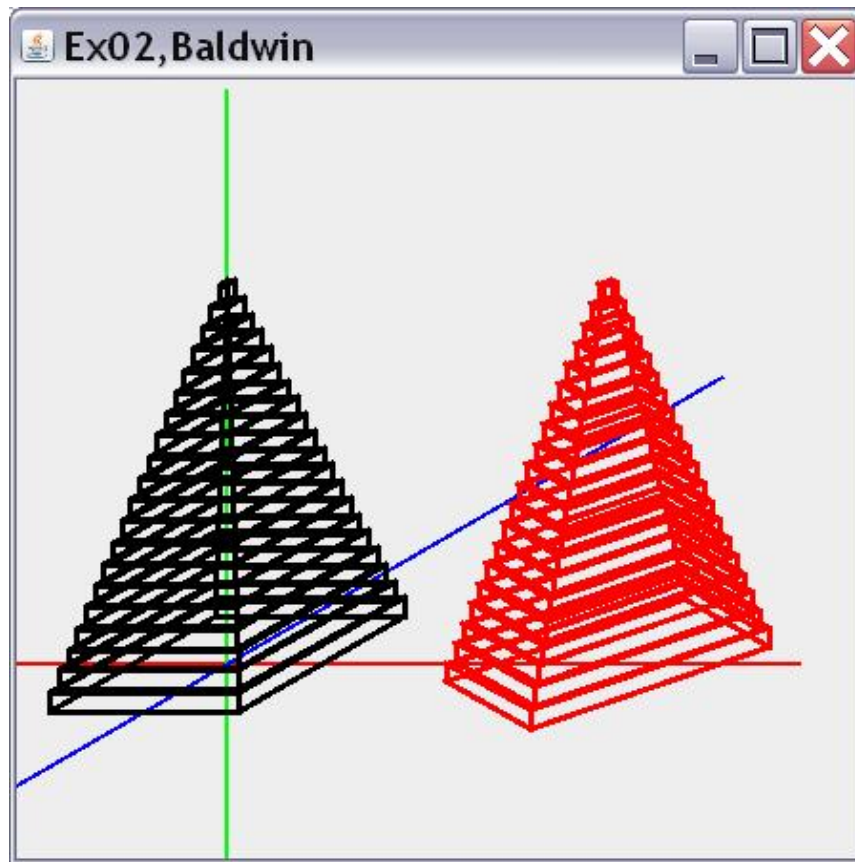
Figure 21 Screen output from Exercise 2.



Exercise 3

Beginning with the two pyramids that you created in [Exercise 2](#), rotate the red pyramid by -30 degrees around an imaginary vertical line at the center of the pyramid as shown in [Figure 22](#).

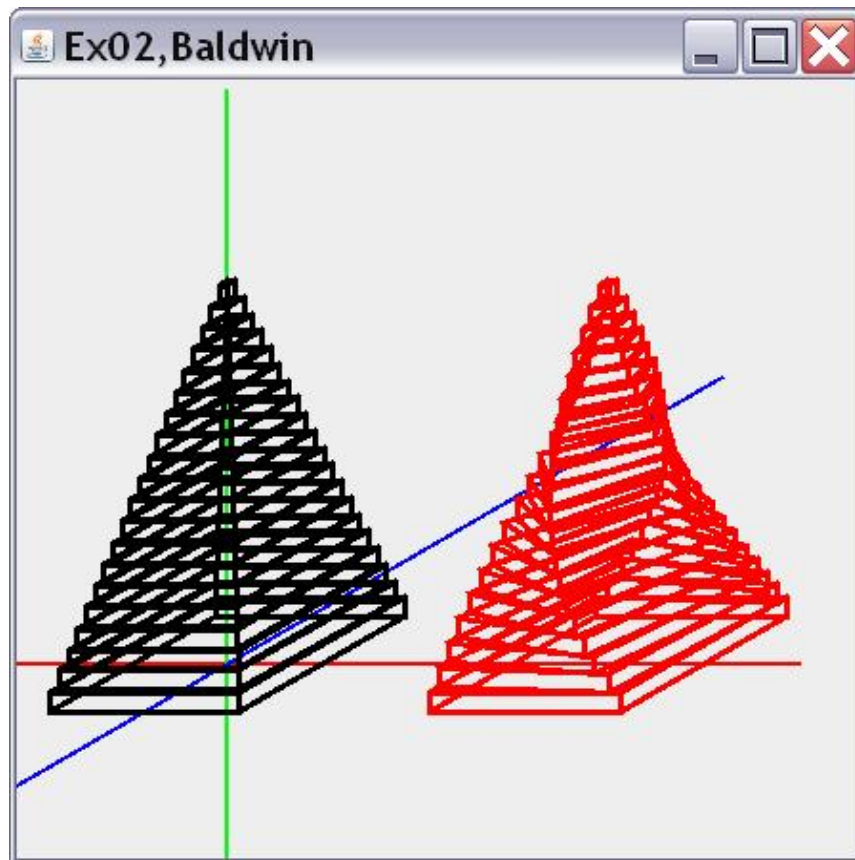
Figure 22 Screen output from Exercise 3.



Exercise 4

Beginning with the two pyramids that you created in [Exercise 2](#), rotate each box around an imaginary vertical line at the center of the pyramid by a negative angle with a progressively greater magnitude so that the rotation of the bottom box is zero and the rotation of the top box is approximately -85 degrees as shown in [Figure 23](#). This produces a 3D object similar to a spiral staircase with the length of each step being less than the length of the step below it.

Figure 23 Screen output from Exercise 4.

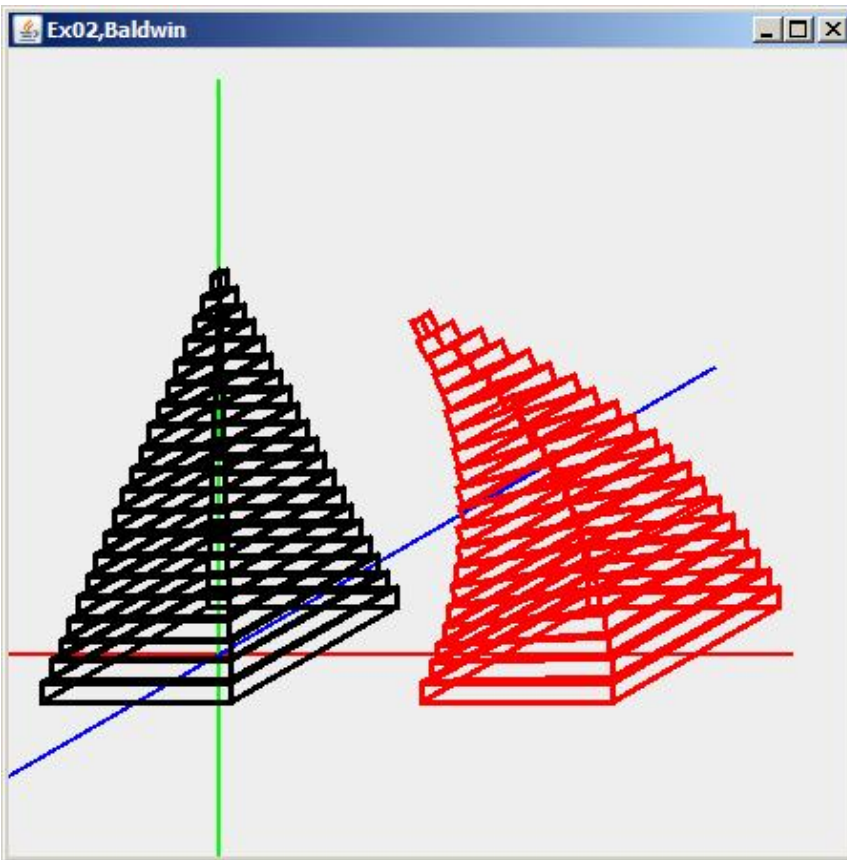


Exercise 5

Beginning with the two pyramids that you created in [Exercise 2](#), rotate each box in the red pyramid around an imaginary line that is parallel to the z-axis and lies in the x-z plane at the center of the bottom box as shown in [Figure 24](#).

Make the rotation angles for each box progressively larger in such a way that the rotation of the bottom box is zero and the rotation of the top box is approximately 28 degrees.

Figure 24 Screen output from Exercise 5.

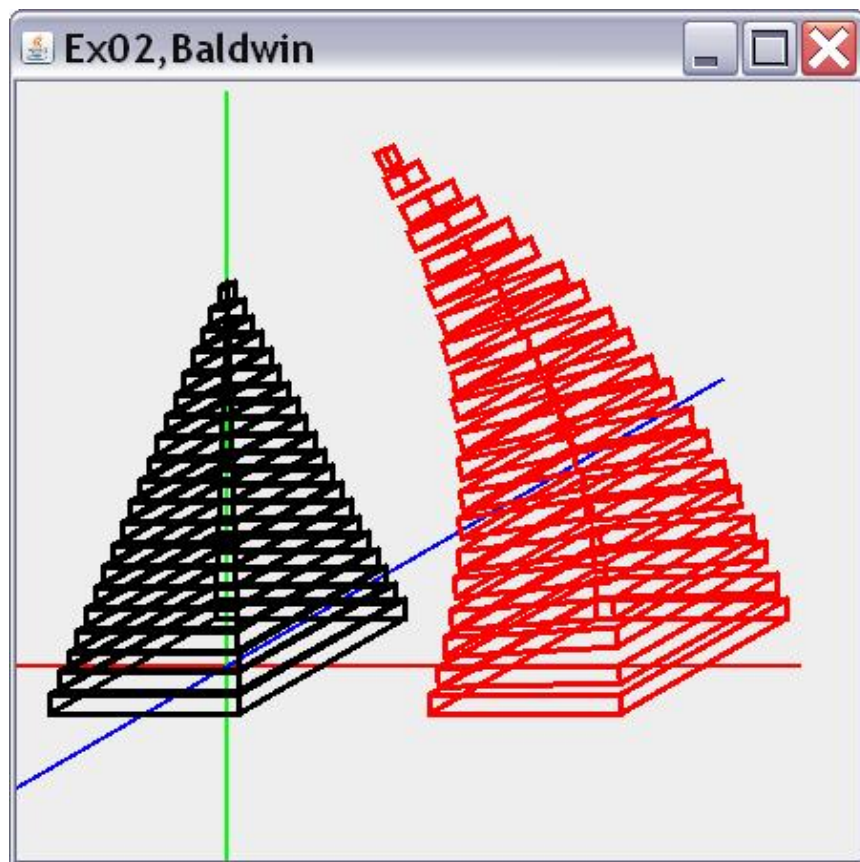


Exercise 6

Beginning with the two pyramids that you created in [Exercise 2](#), rotate each box in the red pyramid around the z-axis as shown in [Figure 25](#).

Make the rotation angles for each box progressively larger in such a way that the rotation of the bottom box is zero and the rotation of the top box is approximately 28 degrees.

Figure 25 Screen output from Exercise 6.



-end-

GAME 2302-0140: Our First 3D Game Program

Learn how to write your first interactive 3D game using the game-math library. Also learn how to write a Java program that simulates flocking behavior such as that exhibited by birds and fish and how to incorporate that behavior in a game.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The game-math library named GM01](#)
 - [The game program named GM01test08](#)
 - [The program named GM01test04](#)
 - [The program named GM01test03](#)
 - [The program named StringArt04](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)
 - [Exercise 3](#)
 - [Exercise 4](#)
 - [Exercise 5](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, you learned how to update the game-math library to support 3D math, how to program the equations for projecting a 3D world onto a 2D plane, how to add vectors in 3D, about scaling, translation, and rotation of a point in both 2D and 3D, about the rotation equations and how to implement them in both 2D and 3D, and much more.

What you will learn

In this module, you will learn how to write your first interactive 3D game using the game-math library named **GM01**. You will also learn how to write a Java program that simulates flocking behavior such as that exhibited by birds and fish and how to incorporate that behavior into a game. Finally, you will examine three other programs that illustrate various aspects of both 2D and 3D animation using the game-math library.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). A school of red prey fish being attacked by a blue predator.
- [Figure 2](#). Screen shot of the graphic output from GM01test04.
- [Figure 3](#). Screen shot of the graphic output from GM01test03.
- [Figure 4](#). Screen shot of the graphic output from StringArt04.

- [Figure 5](#). Prey fish in a large swirling cluster.
- [Figure 6](#). Formation starting to break up due to closeness of the predator.
- [Figure 7](#). 100 prey fish trying to occupy the same location in 3D space.
- [Figure 8](#). 100 prey fish maintaining a reasonable separation in 3D space.
- [Figure 9](#). Twelve predators swimming in a hexagon formation in GM01test04.
- [Figure 10](#). Six predators swimming in a diamond formation.
- [Figure 11](#). Screen output from Exercise 1.
- [Figure 12](#). Screen output from Exercise 4.
- [Figure 13](#). Screen output from Exercise 5.

Listings

- [Listing 1](#). Abbreviated constructor for the GUI class in GM01test08.
- [Listing 2](#). Beginning of the actionPerformed method in GM01test08.
- [Listing 3](#). Code that services the Attack button.
- [Listing 4](#). Code that services the Stop button.
- [Listing 5](#). Beginning of the Animate class and the run method.
- [Listing 6](#). Create the population of prey fish.
- [Listing 7](#). Create and position the predator object.
- [Listing 8](#). A few more housekeeping details.
- [Listing 9](#). Beginning of the animation loop.
- [Listing 10](#). Cause all of the prey fish to spiral toward the center.
- [Listing 11](#). Create a rotated vector.
- [Listing 12](#). Move the prey fish object based on the sum of the vectors.
- [Listing 13](#). Save a clone of the relocated prey fish object.
- [Listing 14](#). Save a normalized direction vector.
- [Listing 15](#). Code for when the population contains only one prey fish.
- [Listing 16](#). Separate the prey fish.
- [Listing 17](#). The remainder of the inner loop.
- [Listing 18](#). Save the primary prey fish object and do another iteration.
- [Listing 19](#). Prey fish objects react to the predator.
- [Listing 20](#). Prey fish object takes evasive action.

- [Listing 21](#). Test the separation again.
- [Listing 22](#). Restore the prey fish to the population.
- [Listing 23](#). Keep the prey fish in the playing field.
- [Listing 24](#). Erase the off-screen image and draw the large circle.
- [Listing 25](#). Draw the prey fish on the off-screen image.
- [Listing 26](#). Cause the predator to slowly circle the cluster of prey fish.
- [Listing 27](#). Execute the attack.
- [Listing 28](#). Draw the predator on the off-screen image.
- [Listing 29](#). Copy off-screen image, insert time delay, etc.
- [Listing 30](#). Source code for the game-math library named GM01.
- [Listing 31](#). Source code for the game program named GM01test08.
- [Listing 32](#). Source code for the program named GM01test04.
- [Listing 33](#). Source code for the program named GM01test03.
- [Listing 34](#). Source code for the program named StringArt04.

Preview

It's time to kick back and have a little fun with what you have learned. In this module, I will present four programs, (*one of which is an interactive 3D game*), that use the current capabilities of the game-math library to produce some fairly serious animation. The programs are named:

- GM01test08
- GM01test04
- GM01test03
- StringArt04

I will explain the game program in detail and will provide a brief description of the other three programs.

GM01test08

The first three programs are based on an [artificial life](#) concept frequently referred to as [boids](#), which is "*a computer model of coordinated animal motion such as bird flocks and fish schools.*"

Note: Flocking fish:

I got the idea for this game while watching a segment of a documentary series named Blue Earth on one of the cable channels. I even surprised myself as to how well I was able to simulate some of the scenes in that TV program using programming concepts similar to those used by Reynolds in boids.

A flocking game program

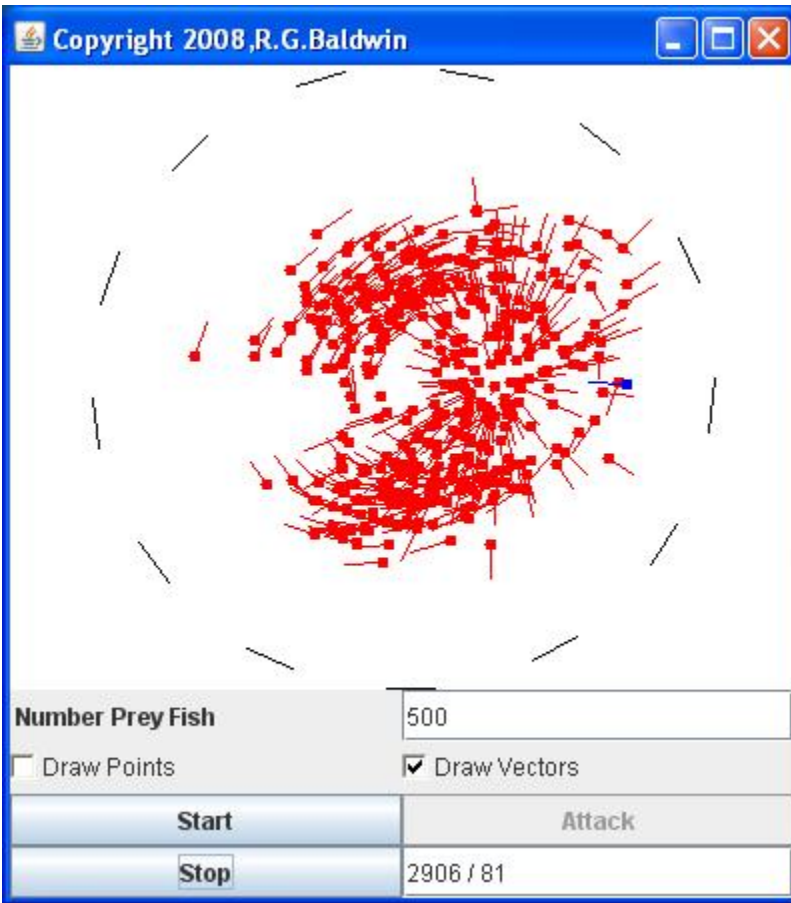
Although the flocking algorithms won't be the same as those used in the original boids program written by Reynolds, the general concepts will be similar.

The first program named **GM01test08** is an interactive 3D game program based on the flocking behavior of fish. This game simulates a school of prey fish being attacked and eaten by a predator. The objective for the player is to cause the predator to catch and eat all of the prey fish in a minimum amount of time.

A preview

I will get into a lot more detail about the game aspects of the program later. For a preview, [Figure 1](#) shows the instant in time when a school of 500 prey fish (*shown in red*) have just been attacked by the blue predator shown near the center right.

Figure 1 A school of red prey fish being attacked by a blue predator.



The predator in [Figure 1](#) has penetrated the school of fish by swimming from left to right, eating fish all along the way, and has emerged from the right side of the school. The prey fish, which were originally in a small tight cluster, have reacted to the presence of the predator by scattering in an attempt to avoid being eaten.

Fully three-dimensional

This game program is fully three-dimensional. Any or all of the fish are capable of swimming in any direction at any time in the 3D world. This is evidenced by the prey fish near the center that appear to have very short or non-existent tails. The fish that appear to have no tails actually have tails that are the same length as the tails on all of the other fish. These fish appear to have no tails because they are swimming either directly toward or directly away from the viewer. I will have more to say about this later.

GM01test04 and GM01test03

These two programs provide essentially the same behavior (*as one another*) , except that one is a 2D program and the other is a 3D program. They are not game programs. Rather, they are demonstration programs that show another aspect of flocking behavior. In these two programs, a large number of predators pursue a single prey object.

A screen shot of the animated output from the 2D version is shown in [Figure 2](#) and a screen shot of the output from the 3D version is shown in [Figure 3](#).

Figure 2 Screen shot of the graphic output from GM01test04.

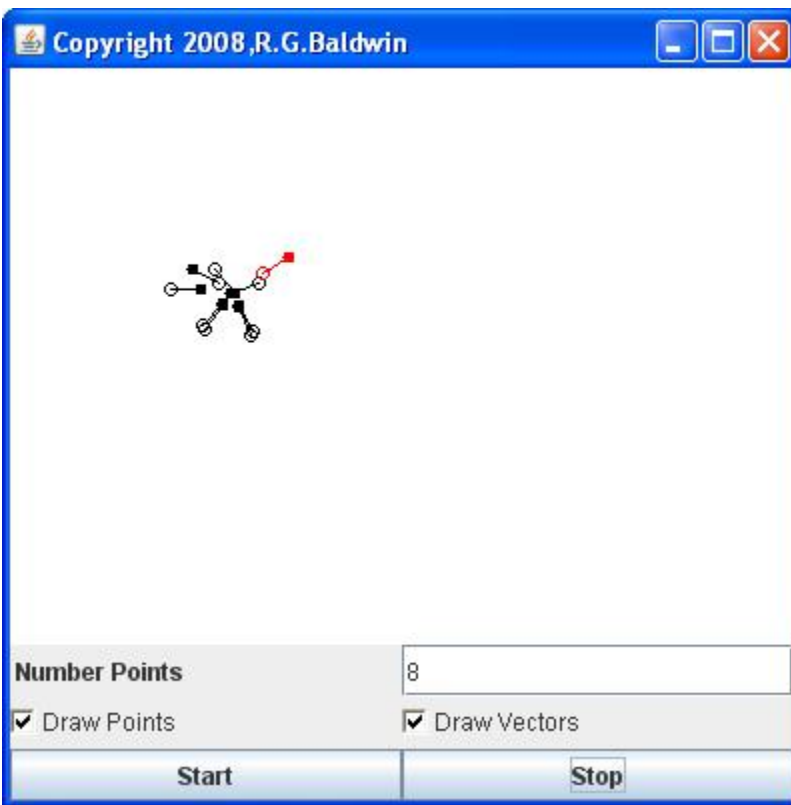


Figure 3 Screen shot of the graphic output from GM01test03.



I included these two programs in this module primarily to provide a good comparison between 2D and 3D, both from a programming viewpoint and a behavior viewpoint.

Eight predator objects and one prey object

In both versions shown in [Figure 2](#) and [Figure 3](#), you see eight predator objects (*colored black*) pursuing a red prey object. The actual length of every predator object is the same as the length of every other predator object and the length of the prey object is the same as the length of the predator objects.

Some significant differences

At first glance, the two versions may look the same. However, they are significantly different. In [Figure 2](#), the prey and the predators are constrained to move in a single plane. Hence, the apparent length of each predator is the same as the length of every other predator and is also the

same as the length of the prey regardless of the direction in which the object is moving.

However, in [Figure 3](#), the predators and the prey are free to move in any direction in a 3D world. As a result, at any point in time, the objects may be moving parallel to or at an angle to the x-y plane. When the 3D world is projected onto the viewing plane, those objects that are moving at an angle to the viewing plane will be foreshortened. In the extreme case, *(as shown earlier in [Figure 1](#))*, when an object is moving directly toward or directly away from the viewer, its apparent length will go to zero. *(At least one predator is almost in that state in [Figure 3](#).)*

In [Figure 3](#) the apparent lengths of the predator and prey objects are all different indicating that they are all moving in slightly different directions in the 3D world.

The flocking algorithms in these two programs are much simpler than the algorithm used in [Figure 1](#). I encourage you to experiment with the flocking algorithms for these two programs to see what you can produce that is new and different from my version of the two programs.

StringArt04

The fourth program named **StringArt04** is completely different from the other three programs. In a previous module, I presented and explained a program that rotates a disk around an arbitrary anchor point in 3D space with the ability to rotate by different angles around each of the three axes. It can be very difficult to visualize exactly what is happening with such complex rotations. This program animates the rotation process and runs it in slow motion so that you can see exactly what is happening as the disk moves and rotates from its initial position and orientation to its final position and orientation.

For example. set the values as follows, click the **Animate** button, and watch the rotations take place.

- Number Points = 24
- Number Loops = 10

- Rotate around Z (deg) = 360
- Rotate around X (deg) = 360
- Rotate around Y (deg) = 360
- X anchor point = 50
- Y anchor point = 0
- Z anchor point = 0

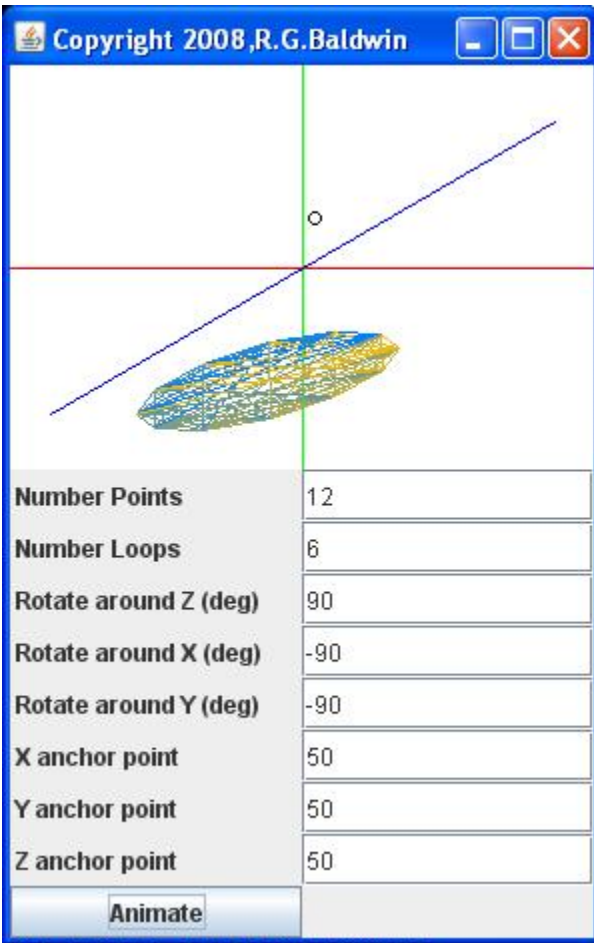
This should give you a good idea as to the process. Then change the parameters to other values, click the button, and observe the results.

It is also useful to set only one of the rotation angles to a non-zero value and watch the rotation around a line parallel to that axis take place.

A stop-action screen shot

[Figure 4](#) shows a screen shot of the disk in motion somewhere between the beginning and the end of its very complex trajectory for one particular set of rotation and anchor point parameters.

Figure 4 Screen shot of the graphic output from StringArt04.



Exercises

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this and previous modules.

Discussion and sample code

The game-math library named GM01

The game-math library named **GM01** has not been modified since I presented and explained it in an earlier module titled [GAME 2302-0135: Venturing into a 3D World](#). A copy of the source code for the library is provided in [Listing 30](#) for your convenience. Since I have explained the

code in the library in earlier modules, I won't repeat those explanations here.

A link to a zip file containing documentation for the library was provided in the [earlier](#) module.

The game program named GM01test08

This is an interactive 3D game program. In fact, this is the first game program that I have presented in this collection on game math. This game simulates a school of prey fish being attacked and eaten by a predator. The objective for the player is to cause the predator to eat all of the prey fish in a minimum amount of time. Although the story line of the game is rather simple, the math involved is not simple at all.

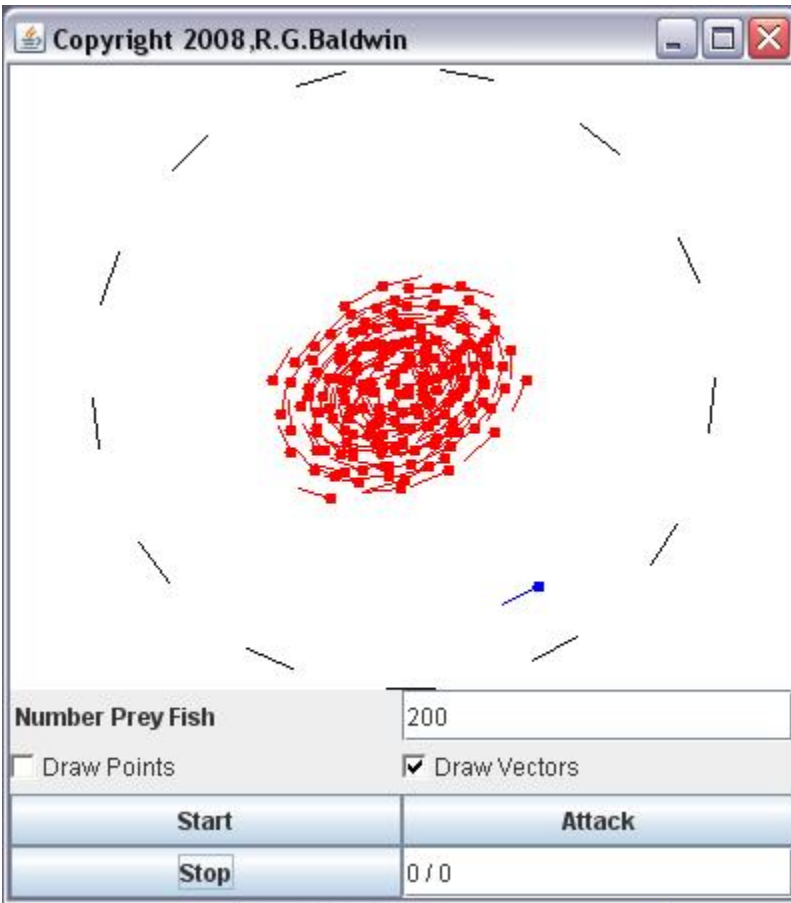
Player initiates attacks

The player in this game initiates a series of attacks by the predator by clicking the **Attack** button shown in [Figure 5](#). The winning strategy for the player is to time the attacks in such a way as to minimize the elapsed time required for the predator to eat all of the prey fish. The final elapsed time depends on both strategy and chance. In other words, as is the case in many games, this game is part strategy and part chance.

A swirling cluster of prey fish

The prey fish, (*shown in red*) , tend to swim in a large swirling cluster as shown in [Figure 5](#) when they aren't being harassed by the predator.

Figure 5 Prey fish in a large swirling cluster.



Predator behavior

The predator, (shown in blue in [Figure 5](#)) , swims around the cluster, darting into and through the cluster when the user clicks the **Attack** button. During an attack, the predator attempts to eat as many prey fish as possible. When the predator attacks, the prey fish tend to scatter (as shown in [Figure 1](#)) , breaking up their tight formation and making it more difficult for the predator to catch them. If the predator is allowed to swim around them again for a short period following an attack, they will form back into a cluster.

Proper timing is important

The attacks should be timed so that the prey fish are in a reasonably tight cluster at the beginning of each attack so that multiple prey fish will be eaten during the attack. However, allowing too much time between attacks is detrimental because it increases the total elapsed time. Thus, the player

must make a tradeoff between elapsed time between attacks and the tightness of the cluster at the beginning of each attack. Another **disadvantage** of waiting too long between attacks will be explained later.

The graphical user interface

In addition to some other controls, the GUI provides an **Attack** button and an *elapsed-time/kill-count* indicator, shown in the bottom right in [Figure 5](#). At any point in time, this indicator displays the elapsed time in milliseconds when the most recent prey fish was eaten along with the total number of prey fish that have been eaten. The two values are separated by a "/" character.

When all of the prey fish have been eaten by the predator, the elapsed time indicator shows the time in milliseconds required for the predator to catch and eat all of the prey fish. It also shows the number of prey fish that were eaten, which should match the value that the player entered into the upper-right text field before starting the game. (*This field contains 500 in [Figure 1](#) and 200 in [Figure 5](#)*).

Not initially in attack mode

When the player clicks the **Start** button and the game begins, the predator is not in attack mode. Rather, the predator swims around the school of prey fish encouraging them to bunch up into a smaller and tighter cluster as shown in [Figure 5](#). The purpose of this behavior is to increase the number of fish that will be eaten when an attack is made.

The attack will be more or less successful

This circling behavior on the part of the predator continues until the player clicks the **Attack** button, at which time the predator enters the attack mode and makes an attack on the cluster as shown in [Figure 1](#). If the player clicks the **Attack** button too early, or doesn't wait long enough between attacks, the prey fish will be in a loose cluster and the predator will eat very few fish during the attack.

The predator always has an impact

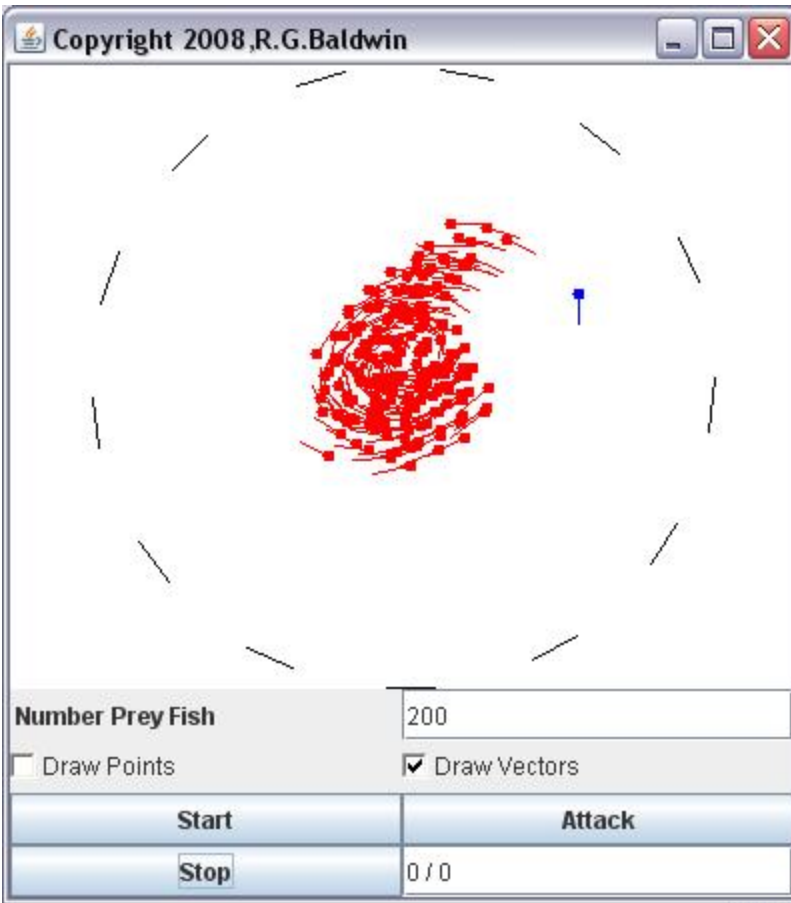
Even when the predator is not in attack mode, its presence has a significant effect on the behavior of the school of prey fish. As the predator swims around the prey fish, they tend to bunch up into a smaller and tighter cluster, but when the predator swims too close, the prey fish tend to panic and scatter, breaking up the tight cluster as shown in [Figure 6](#).

Therefore, if the player waits too long to click the **Attack** button or waits too long between attacks, the predator will have spiraled in so close to the prey fish that they will break formation and begin to scatter, making it difficult for the predator to eat a large number of fish during the next attack. This is the other disadvantage of waiting too long to attack that I mentioned [earlier](#).

Formation starting to break up due to closeness of the predator

[Figure 6](#) shows the formation starting to break up because the predator has strayed too close to the cluster of prey fish. *(The fish in the upper right of the formation have pulled out of the cluster and have started to flee the predator.)*

Figure 6 Formation starting to break up due to closeness of the predator.



An effective defense mechanism

The prey fish have a fairly effective defense mechanism and can do a reasonably good job of escaping the predator. The predator can increase the odds of catching the prey fish by attacking very fast. (*You can modify the code that controls the speed of the attack.*) If the predator goes into the formation slowly, the prey fish will simply run away as is starting to be the case in [Figure 6](#).

Note: Cleaning up the leftovers:

The final three or four prey fish are often the most difficult to catch because they have more room to run without colliding with another fish.

Captured fish are removed from the population

When the predator is successful in eating a prey fish, that fish is removed from the prey-fish population causing the prey-fish population to decrease over time. Also, each time the predator eats a prey fish, the program emits an audible beep to provide feedback to the player.

The other GUI components

In addition to the elapsed time indicator and the **Attack** button, the GUI contains an input text field by which the player can specify the number of prey fish that will be in the population when the game begins. The GUI also contains a **Start** button and a **Stop** button. Finally, the GUI contains check boxes that allow the player to display points only, direction vectors only, or both for the prey fish. *(Only the direction vector is displayed for the predator.)*

Playing the game

In practice, the player specifies the number of randomly-placed prey fish that will be in the population and clicks the **Start** button to start the game. At this point, the prey fish swim in a large swirling cluster and the predator swims around them encouraging them to form a tighter cluster as shown in [Figure 5](#).

Prey fish motion is random but each prey fish generally tends to spiral toward the center of the cluster. When the user clicks the **Attack** button, the predator turns and swims rapidly into and through the center of the cluster of prey fish. If the predator manages to get to within a specified distance from a prey fish, that prey fish will be deemed to have been eaten, and will be removed from the population. However, each prey fish will sense that the predator is coming and will try to escape. Whether or not an individual prey fish manages to escape the predator when an encounter between the two occurs is based on a random number generator.

The Start and Stop buttons

The animation continues until all of the fish have been eaten or the user clicks the **Stop** button. The user can click the **Stop** button at any time, change any of the parameters, and then click the **Start** button again to restart the game with zero elapsed time and different parameters.

Displaying the vectors

The animation is most impressive when the direction vectors are displayed for the prey fish because the vectors provide a lot of visual information about how the prey fish are reacting to the predator.

A spherical playing-field boundary

In addition to the school of prey fish and the predator, the graphical output also shows a large circle drawn with broken lines, as shown in [Figure 5](#). This circle represents the intersection of a sphere and the x-y plane.

The purpose of the sphere, which is centered on the origin, is to provide a soft boundary for keeping the prey fish and the predator inside the 3D playing field. The prey fish and the predator all have a tendency to remain inside the sphere, but they may occasionally stray outside the sphere. If they do, the program code will encourage them to return to the 3D playing field inside the sphere.

A 3D display

As mentioned earlier, this game is fully three dimensional. The prey fish and the predator are free to swim in any direction in 3D space. The tails on the prey fish and the predator appear longest when they are swimming parallel to the x-y plane. As they change their angle relative to the x-y plane, the tails appear to become shorter or longer and in some cases, they appear not to have a tail at all. The fish that appear to have no tails are either swimming directly toward the viewer or swimming directly away from the viewer.

This can best be observed by clicking the **Stop** button just as the fish scatter during an attack and freezing the state of the fish in the 3D world as shown in [Figure 1](#). If you examine the image at that point, you are likely to see

some fish with shorter tails than other fish. This is evident by some of the prey fish near the center of [Figure 1](#) that appear to have no tails at all.

Enough talk, let's see some code

A complete listing of this program is provided in [Listing 31](#) near the end of the module.

Portions of this program are very similar to programs that I have explained in earlier modules in this series. I won't repeat those explanations here. Rather, I will concentrate mostly on the code that is new and different in this module.

Abbreviated constructor for the GUI class

[Listing 1](#) shows an abbreviated constructor for the GUI class. *(Much of the code in the constructor has been explained before, and was deleted from Listing 1 for brevity.)*

Listing 1 . Abbreviated constructor for the GUI class in GM01test08.

Listing 1 . Abbreviated constructor for the GUI class in GM01test08.

```
GUI(){//constructor

//Code deleted for brevity

    //Register this object as an action
listener on all
    // three buttons.
    startButton.addActionListener(this);
    attackButton.addActionListener(this);
    stopButton.addActionListener(this);

    //Make the drawing color RED at startup.
    g2D.setColor(Color.RED);

} //end constructor
```

If you examine [Listing 31](#), you will see that the **GUI** class implements the **ActionListener** interface. Therefore, an object of the **GUI** class can be registered as an action listener on a button. [Listing 1](#) registers the object of the **GUI** class as a listener on all three of the buttons shown in [Figure 6](#).

Beginning of the actionPerformed method

The **actionPerformed** method that begins in [Listing 2](#) is called whenever the player clicks any of the three buttons shown in [Figure 6](#).

Listing 2 . Beginning of the actionPerformed method in GM01test08.

```
public void actionPerformed(ActionEvent e){

    if(e.getActionCommand().equals("Start") &&
!animate){
        //Service the Start button.

        //Get several user input values.
        numberPoints = Integer.parseInt(
numberPointsField.getText());

        if(drawPointsBox.getState()){
            drawPoints = true;
        }else{
            drawPoints = false;
        }//end else

        if(drawVectorsBox.getState()){
            drawVectors = true;
        }else{
            drawVectors = false;
        }//end else

        //Initialize some working variables used
in the
        // animation process.
        animate = true;
        baseTime = new Date().getTime();
        killCount = 0;

        //Enable the Attack button.
        attackButton.setEnabled(true);
```


Listing 2 . Beginning of the actionPerformed method in GM01test08.

```
        //Initialize the text in the timer
        field.
        timer.setText("0 / 0");

        //Cause the run method belonging to the
        animation
        // thread object to be executed.
        new Animate().start();
    }//end if
```

Servicing the Start button

The code in [Listing 2](#) is executed when the user clicks the **Start** button. All of this code is also very similar to code that I have explained in previous modules, so further explanation beyond the embedded comments should not be necessary. However, I will emphasize two statements in [Listing 2](#).

The first statement that I will emphasize (*about two-thirds of the way down*) enables the **Attack** button. The second statement (*at the end*) instantiates a new object of the **Thread** class named **Animate** and causes the **run** method of the animation thread to be executed. I will explain the animation thread later.

Code that services the Attack button

The code in [Listing 3](#) is executed when the player clicks the **Attack** button after it is enabled by the code in [Listing 2](#).

Listing 3 . Code that services the Attack button.

```
        if(e.getActionCommand().equals("Attack")
        && animate){

            attack = true;

            predatorVec =

predator.getDisplacementVector(preycenter).

scale(1.0);

            //Disable the Attack button. It will be
enabled
            // again when the attack is finished.
            attackButton.setEnabled(false);
        }//end if
```

[Listing 3](#) begins by setting the value of the variable named **attack** to true. As you will see later, this causes the predator to attack the prey fish in the animation loop.

Control the speed and direction of the predator attack

Then [Listing 3](#) calls the **getDisplacementVector** method and the **scale** method of the game-math library to set the value of the variable named **predatorVec** to a direction that will be used to point the predator toward the center of the prey-fish cluster. In [Listing 3](#), the scale factor is 1.0, so the application of the scale factor does nothing. However, you can modify this value to control the speed of the predator during the upcoming attack. If you increase the scale factor, the predator will move faster. If you decrease the scale factor, the predator will move more slowly.

Temporarily disable the Attack button

Finally, [Listing 3](#) temporarily disables the **Attack** button so that it will have no effect if the player clicks it again during the attack. You could also remove this statement, slow down the attack, and experiment with multiple clicks on the **Attack** button during the course of an attack to see what happens.

Code that services the Stop button

The code in [Listing 4](#) is executed when the player clicks the **Stop** button.

Listing 4 . Code that services the Stop button.

```
        if(e.getActionCommand().equals("Stop") &&
animate){

            //This will cause the run method to
            terminate and
            // stop the animation. It will also
            clear the
            // attack flag so that the next time the
            animation
            // is started, the predator won't be in
            the
            // attack mode.
            animate = false;
            attack = false;
        }//end if

    }//end actionPerformed
```

No explanation beyond the embedded comment should be required for this code.

Beginning of the **Animate** class and the **run** method

When the **Start** button is clicked, the code in [Listing 2](#) instantiates a new object of the **Thread** class named **Animate** and causes the **run** method belonging to that object to be executed. [Listing 5](#) shows the beginning of the **Animate** class and its **run** method.

Listing 5 . Beginning of the **Animate** class and the **run** method.

```
class Animate extends Thread{
    //Declare a general purpose variable of
    type Point3D.
    // It will be used for a variety of
    purposes.
    GM01.Point3D tempPrey;

    //Declare two general purpose variables of
    type
    // Vector3D. They will be used for a
    variety of
    // purposes.
    GM01.Vector3D tempVectorA;
    GM01.Vector3D tempVectorB;
    //-----
    -----//

    public void run(){
        //This method is executed when start is
        called on
```

Listing 5 . Beginning of the Animate class and the run method.

```
// this Thread object.  
//Create a new empty container for the  
prey objects.  
//Note the use of "Generics" syntax.  
preyObjects = new  
ArrayList<GM01.Point3D>();  
  
//Create a new empty container for the  
vectors. The  
// only reason the vectors are saved is  
so that they  
// can be displayed later  
displayVectors = new  
ArrayList<GM01.Vector3D>();
```

Importance of the game-math library named GM01

The code in [Listing 5](#) is straightforward and shouldn't require an explanation beyond the embedded comments. However, I will emphasize the heavy use of the game-math library classes named **GM01.Point3D** and **GM01.Vector3D** in the code in [Listing 5](#).

This entire game program is heavily dependent on the use of the **ColMatrix3D** , **Point3D** , **Vector3D** , and **Line3D** classes along with some of the static methods in the game-math library named **GM01** . If you were to start from scratch and write this game program without the availability of the game-math library, the program would be much longer and would be much more complex.

Create the population of prey fish

[Listing 6](#) creates the population of prey fish and positions them at random locations in 3D space. [Listing 6](#) also stores references to the prey fish objects in the **preyObjects** container, which was created in [Listing 5](#).

Listing 6 . Create the population of prey fish.

```
        for(int cnt = 0;cnt <
numberPoints;cnt++){
            preyObjects.add(new GM01.Point3D(
                new GM01.ColMatrix3D(
                    100*
(random.nextDouble()-0.5),
                    100*
(random.nextDouble()-0.5),
                    100*
(random.nextDouble()-0.5))));

            //Populate the displayVectors
collection with
            // dummy vectors.
            displayVectors.add(tempVectorA);
        }//end for loop
```

In addition, [Listing 6](#) populates a container named **displayVectors** that was also created in [Listing 5](#) to store a direction vector belonging to each prey fish. This container is populated with dummy vectors in [Listing 6](#), simply to set the size of the container to be the same as the size of the container containing references to the prey fish objects. *(There is probably a more efficient way to set the size of that container.)*

Note that each prey fish object is actually represented by an object of the game-math library class named **GM01.Point3D** .

Create and position the predator object

[Listing 7](#) creates an object of the **GM01.Point3D** class that will represent the predator in the game.

Listing 7 . Create and position the predator object.

```
predator = new GM01.Point3D(  
    new  
    GM01.ColMatrix3D(-100,100,-100));
```

The initial position given to the predator in [Listing 7](#) causes it to appear in the game near the top center of the screen when the user clicks the **Start** button. You could also make the initial position of the predator random if you think that would improve the game.

A few more housekeeping details

[Listing 8](#) takes care of a few more housekeeping details before the animation actually begins.

Listing 8 . A few more housekeeping details.

Listing 8 . A few more housekeeping details.

```
        //Create a reference point to mark the
origin. Among
        // other things, it will be used as the
center of a
        // sphere, which in turn will be used to
attempt to
        // keep the prey and predator objects
from leaving
        // the playing field.
        GM01.Point3D origin =
            new GM01.Point3D(new
GM01.ColMatrix3D(0,0,0));

        //Declare some variables that will be
used to
        // compute and save the average position
of all of
        // the prey objects.
        double xSum = 0;
        double ySum = 0;
        double zSum = 0;
```

Let the show begin

[Listing 9](#) shows the beginning of the animation loop, which will continue to execute for as long as the value of the variable named **animate** is true.

Listing 9 . Beginning of the animation loop.

```
        while(animate){
            //Compute and save the average
            position of all the
            // prey objects at the beginning of
            the loop. Save
            // the average position in the Point3D
            object
            // referred to by preyCenter.
            xSum = 0;
            ySum = 0;
            zSum = 0;

            for(int cnt = 0;cnt <
            preyObjects.size();cnt++){
                tempPrey = preyObjects.get(cnt);
                xSum += tempPrey.getData(0);
                ySum += tempPrey.getData(1);
                zSum += tempPrey.getData(2);
            }//end for loop

            //Construct a reference point at the
            average
            // position of all the prey objects.
            preyCenter = new GM01.Point3D(
                new
                GM01.ColMatrix3D(xSum/preyObjects.size(),
                ySum/preyObjects.size(),
                zSum/preyObjects.size()));
```

The value of **animate** is set to true by clicking the **Start** button and is set to false by clicking the **Stop** button. Setting the value to false causes the **run** method to terminate, thereby stopping the animation.

Attack toward the geometric center

When the predator attacks, it will move to and through the geometric center of the cluster of prey fish, eating prey fish all along the way. The geometric center of the cluster of prey fish is computed and saved by the code in [Listing 9](#) as the average location of all the prey fish in the population.

Note that a vector that points to the geometric center may or may not be a good indicator of the best direction for eating large numbers of prey fish during an attack. In [Figure 1](#), for example, a predator attacking from the 11:00 o'clock position and heading toward the center would encounter quite a few prey fish. However, a predator attacking from the 8:00 o'clock position and heading toward the center would encounter far fewer prey fish.

You will also learn that although the predator will always attack in a direction pointing toward this geometric center, the positions of the prey fish can change after the center is computed and before the attack begins, causing the position of the actual geometric center to change, before the predator has an opportunity to attack. That can decrease the probability of a successful attack by the predator.

These issues form part of the strategy of the game that must be mastered by the player in order to earn a good score.

Cause all of the prey fish to spiral toward the center

[Listing 10](#) shows the beginning of a fairly complicated algorithm that causes all of the prey fish to have a tendency to spiral toward the center of the cluster of prey fish.

Listing 10 . Cause all of the prey fish to spiral toward the center.

```
        for(int cnt = 0;cnt <
preyObjects.size();cnt++){
            if(preObjects.size() > 1){
                //Get the next prey object
                tempPrey = preyObjects.get(cnt);
                //Find the displacement vector
from this prey
                // object to the preyCenter
                tempVectorA =
tempPrey.getDisplacementVector(
preyCenter);
```

The code in the body of the **for** loop that begins in [Listing 10](#) is executed once during each animation cycle for each of the prey fish in the population of prey fish.

To begin with, the **if** statement prevents the tendency to spiral toward the geometric center from being applied when there is only one prey fish remaining in the population. I will leave it as an exercise for the student to think about this and to decide whether or not this is a good decision.

The code in [Listing 10](#) gets a reference to the next prey fish object in the population and then gets and saves a reference to a displacement vector pointing from the prey fish to the geometric center.

Create a rotated vector.

[Listing 11](#) creates, scales, and saves a new **GM01.Vector3D** object having the same component values as the displacement vector from [Listing 11](#), but assigning those values to different axes.

Listing 11 . Create a rotated vector.

```
tempVectorB = new GM01.Vector3D(  
    new GM01.ColMatrix3D(  
        tempVectorA.getData(1),  
        tempVectorA.getData(2),  
        tempVectorA.getData(0))) .scale(1.2);
```

Once again, I will leave it as an exercise for the student to think about the length and direction of such a vector in relation to the original displacement vector. It might help to begin by drawing some vectors in 2D that swap components and then extending the same thinking to 3D. One thing is for certain, unless all of the components have the same value, the new vector created in [Listing 11](#) has a different length and direction than the displacement vector created in [Listing 10](#), and that would be true even if the vector from [Listing 11](#) had not been scaled.

Move the prey fish object based on the sum of the vectors

[Listing 12](#) begins by adding the two vectors that were created in [Listing 10](#) and [Listing 11](#), scaling the vector from [Listing 10](#) before performing the addition. Note that the scale factors applied to the rotated vector in Listing 11 is different from the scale factor applied to the original displacement vector in [Listing 12](#). The ratio of these two scale factors influences the tendency of the prey fish object to move toward the center relative to its tendency to move around the center.

Listing 12 . Move the prey fish object based on the sum of the vectors.

```
tempVectorA =  
tempVectorA.scale(0.9).add(tempVectorB);  
tempPrey =  
tempPrey.addVectorToPoint(  
tempVectorA.scale(0.1));
```

Then [Listing 12](#) calls the **GM01.Point3D.addVectorToPoint** method to relocate the prey fish object to a new position based on the scaled sum of the two vectors. The bottom line is that this will cause the prey fish object to spiral toward the geometric center that was computed in [Listing 9](#) as the animation progresses.

Another scale factor is applied to the sum vector before using it to relocate the prey fish object. This scale factor controls the overall speed of the prey fish. Increasing the scale factor causes the prey fish to spiral faster. *(Increasing the scale factor also causes some interesting patterns to appear and if you increase it too much, the prey fish will all leave the playing field.)*

Save a clone of the relocated prey fish object

[Listing 13](#) creates a clone of the relocated prey fish object and uses it to replace the original prey fish object in the container.

Listing 13 . Save a clone of the relocated prey fish object.

```
preyObjects.set(cnt, tempPrey.clone());
```

Creating and saving a reference to a clone instead of saving a reference to the relocated prey fish object may be overkill. However, I simply wanted to guard against the possibility of ending up with a corrupted object later due to the repeated use of the reference variable. I will leave it up to the student to think about this and to decide if this was a good or a bad decision.

Save a normalized direction vector

[Listing 14](#) calls the **GM01.Vector3D.normalize** method to create a vector having a length of 15 units and the same direction as the vector that was used to relocate the prey fish object.

Listing 14 . Save a normalized direction vector.

```
displayVectors.set(  
cnt, tempVectorA.normalize().scale(15.0));
```

This vector is saved and used for drawing later. Drawing this vector in conjunction with the point that represents the prey fish object produces the tails shown on the prey fish objects in [Figure 1](#).

Code for when the population contains only one prey fish

[Listing 15](#) shows the **else** clause that matches up with the **if** statement in [Listing 10](#). This code is executed when the population has been reduced to only one prey fish object.

Listing 15 . Code for when the population contains only one prey fish.

```
        }else{
            displayVectors.set(
                cnt,new GM01.Vector3D(
                    new
GM01.ColMatrix3D(10,10,0)));
        }//end else

    }//end loop to spiral prey objects
    toward center.
```

[Listing 15](#) saves a dummy direction vector for drawing later. This is necessary to prevent a null pointer exception when the user specifies only one prey fish object and then clicks the **Start** button.

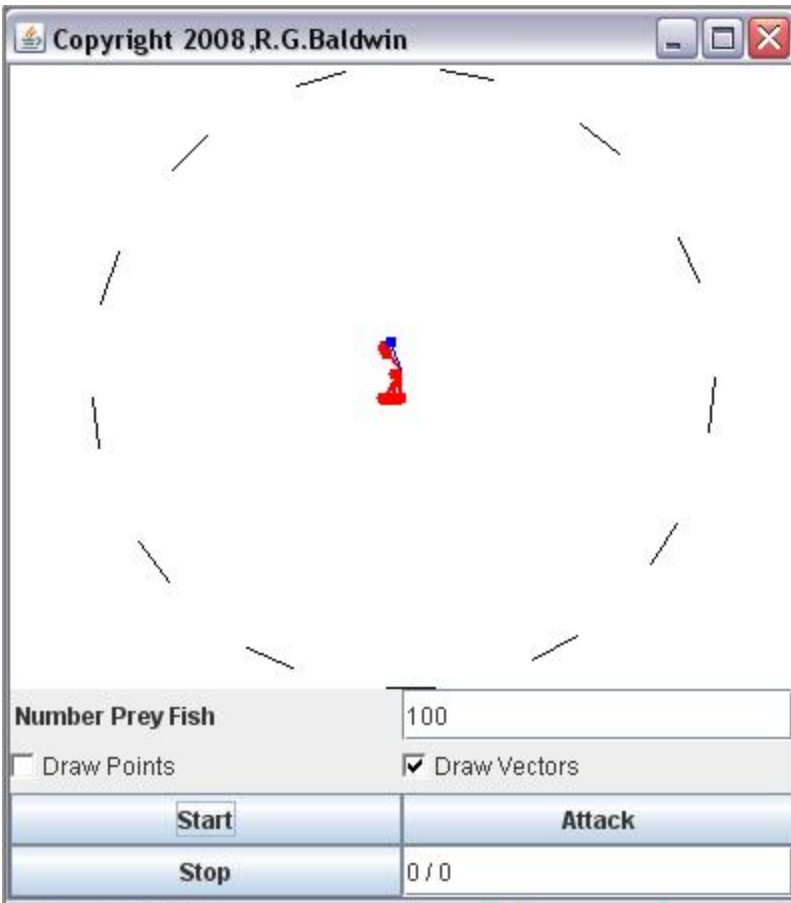
Not the end of the story

[Listing 15](#) also signals the end of the **for** loop that causes the prey fish to spiral toward the center. The code in this **for** loop is sometimes referred to as a *cohesion* algorithm in flocking terminology. It causes the prey fish to stay together as a group.

This is not the end of the story, however. If the code in this cohesion algorithm were the only code controlling the behavior of the prey fish in this animation, they would simply continue spiraling toward the center,

eventually producing a cluster that looks something like that shown in [Figure 7](#) where 100 prey fish are trying to occupy the same location in 3D space.

Figure 7 100 prey fish trying to occupy the same location in 3D space.



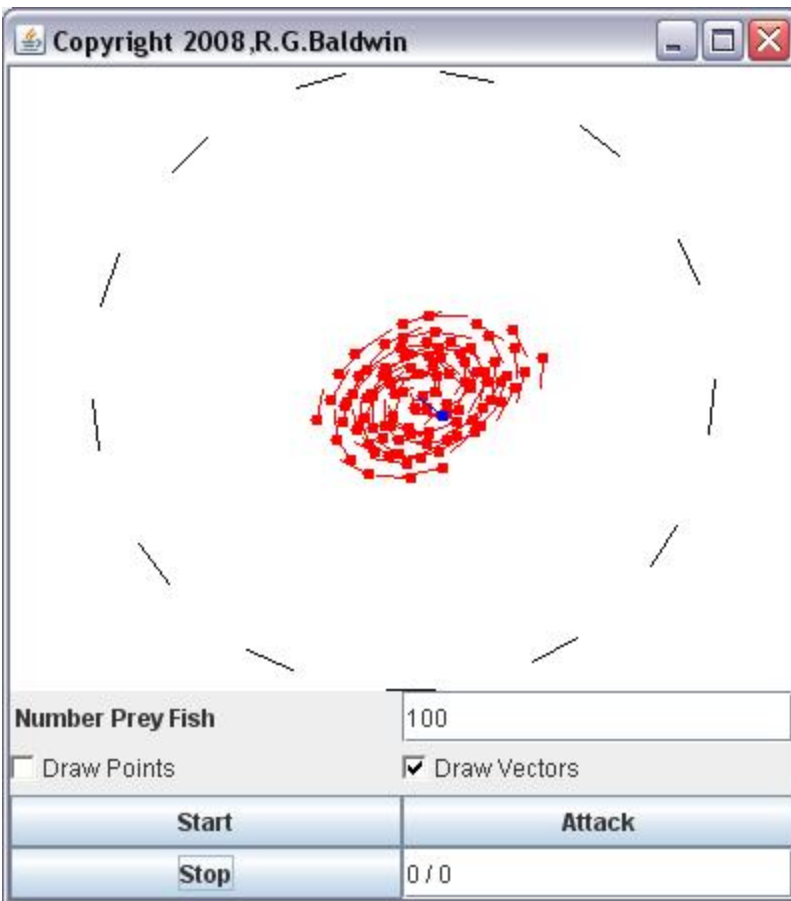
Interesting but also boring

While it may be interesting to watch the animation progress to this point, the animation becomes very boring when all of the prey fish cluster at the center. What we need is an additional algorithm that will cause each prey fish to attempt to maintain a respectable distance between itself and the other prey fish.

Prey fish maintaining a reasonable separation

For example, [Figure 8](#) shows the result of temporarily making each prey fish immune to the presence of the predator, telling each prey fish to spiral toward the center, and also telling each prey fish to maintain a separation of ten units (*pixels*) between itself and all of the other prey fish.

Figure 8 100 prey fish maintaining a reasonable separation in 3D space.



A 3D world

Remember, [Figure 8](#) is a 3D world projected onto on a 2D screen display. Even if every prey fish is separated from every other prey fish by at least ten pixels (*which is probably not the case as I will explain later*) , the projection of the 3D world onto the 2D display can make it appear that two or more prey fish occupy the same location.

Separate the prey fish

[Listing 16](#) shows the beginning of a pair of nested **for** loops that attempt to keep the prey fish from colliding with one another by moving each prey fish object away from its close neighbors if necessary. In flocking terminology, this is often referred to as a *separation* algorithm.

Listing 16 . Separate the prey fish.

```
GM01.Point3D refPrey = null;
GM01.Point3D testPrey= null;
for(int row = 0;row <
preyObjects.size();row++){
    refPrey = preyObjects.get(row);
    //Compare the position of the
reference prey
    // object with the positions of each
of the
        // other prey objects.
        for(int col = 0;col <
preyObjects.size();col++){
            //Get another prey object for
proximity test.
            testPrey = preyObjects.get(col);
```

This algorithm gets a reference to each prey fish object (*primary object*) and compares its position with the positions of all the other prey fish objects (*secondary objects*) . If the primary object is too close to a secondary object, the primary object is moved away from the secondary object.

Not a perfect algorithm

This is not a perfect algorithm however. A primary object can be moved away from all of its neighbors early in the execution of the algorithm, but a neighbor could be moved closer to the primary object later in the execution of the algorithm. While not perfect, the algorithm does a pretty respectable job of keeping the prey fish separated as evidenced by comparing the positions of the prey fish in [Figure 7](#) with the positions of the prey fish in [Figure 8](#). This *separation* algorithm was disabled in [Figure 7](#) and was enabled in [Figure 8](#).

Gets two objects that will be compared

[Listing 16](#) gets a reference to a primary prey fish object in the outer loop that iterates on the counter named **row** and gets a reference to one of the secondary prey fish objects to which it will be compared in the inner loop that iterates on the counter named **col**.

The remainder of the inner loop

[Listing 17](#) shows the remainder of the inner loop.

Listing 17 . The remainder of the inner loop.

Listing 17 . The remainder of the inner loop.

```
        //Don't test a prey object against
    itself.
        if(col != row){
            //Get the vector from the
    refPrey object to
            // the testPrey object.
            tempVectorA = refPrey.

    getDisplacementVector(testPrey);

            //If refPrey is too close to
    testPrey, move
            // it away from the testPrey
    object.
            if(tempVectorA.getLength() < 10)
    {
                //Move refPrey away from
    testPrey by a
                // small amount in the
    opposite direction.
                refPrey =
    refPrey.addVectorToPoint(
    tempVectorA.scale(0.2).negate());

                }//end if on proximity test
            }//end if col != row
        }//end loop on col
```

It wouldn't make any sense to compare the position of a prey fish with itself, so the **if** statement in [Listing 17](#) prevents that from happening.

Code is relatively straightforward

Since you already know how to use most of the methods in the game-math library, you should have no difficulty understanding the code in [Listing 17](#). This code:

- gets the displacement vector that defines the separation between the primary object and the secondary object,
- compares the length of that vector with 10 units, and
- moves the primary prey fish object in the opposite direction by 20-percent of the length of the separation vector if the separation is less than 10 units.

As mentioned earlier, however, moving the primary fish object away from one prey fish could cause it to be moved closer to a different prey fish object, so the separation algorithm is far from perfect.

[Listing 17](#) signals the end of the inner loop that began in [Listing 16](#).

Save the primary prey fish object and do another iteration

The primary prey fish object may or may not have been moved. Regardless, a clone of that object is created and saved in the container that contains the prey-fish population in [Listing 18](#).

Listing 18 . Save the primary prey fish object and do another iteration.

```
preyObjects.set(row,refPrey.clone());  
    }//end loop on row
```

After the object is saved in [Listing 18](#), control is transferred back to the top of the outer **for** loop in [Listing 16](#) and the next prey fish object in the population is compared with all of the other prey-fish objects in the population, making corrections to the position of the prey fish as necessary.

Prey fish objects react to the predator

As I mentioned earlier, each prey fish object has a reasonably good defense mechanism to avoid being eaten by the predator. However, in the final analysis, whether or not a prey fish will escape when it encounters the predator face to face is a random process. To some extent, success or failure to escape depends on how quickly the prey fish senses the presence of the predator.

[Listing 19](#) is the beginning of a **for** loop in which the proximity of each prey fish to the predator is tested and evasive action on the part of the prey fish is taken if the distance between the two is less than 50 units.

Listing 19 . Prey fish objects react to the predator.

Listing 19 . Prey fish objects react to the predator.

```
        for(int cnt = 0;cnt <
preyObjects.size();cnt++){
            tempPrey = preyObjects.get(cnt);

            //Get a displacement vector from the
prey object
            // to the predator.
            tempVectorA =
tempPrey.getDisplacementVector(

predator);
```

The code in [Listing 19](#) gets the next prey fish in the population and gets a displacement vector describing the separation of the prey fish from the predator.

Prey fish object takes evasive action

If the prey fish is at least 50 units away from the predator, the prey fish simply goes on swimming without concern for the predator. However, if the prey fish is less than 50 units away from the predator, the prey fish takes evasive action in an attempt to escape the predator. The evasive action is accomplished in [Listing 20](#), and this is where some of the random behavior comes into play.

Listing 20 . Prey fish object takes evasive action.

Listing 20 . Prey fish object takes evasive action.

```
        if(tempVectorA.getLength() < 50){
            tempVectorA =
tempVectorA.negate().scale(
random.nextDouble());
            tempPrey =
tempPrey.addVectorToPoint(tempVectorA);
```

The prey fish moves by a random distance

In [Listing 20](#), the negative of the displacement vector separating the prey fish from the predator is scaled by a random value ranging from 0 to 1. Then the prey fish is moved by the distance and direction specified by the resulting vector. If the random value is 0, the prey fish won't be moved at all. If the random value is 1, the distance between the prey fish and the predator will be doubled. For values between 0 and 1, the prey fish will be moved different distances away from the predator.

Did the prey fish escape?

Once the prey fish has taken the evasive action and has (*possibly*) moved away from the predator, the separation between the prey fish and the predator is tested again in [Listing 21](#).

Listing 21 . Test the separation again.

Listing 21 . Test the separation again.

```
tempVectorA =  
tempPrey.getDisplacementVector(predator);  
  
    if(tempVectorA.getLength() < 25){  
        tempPrey =  
preyObjects.remove(cnt);  
        displayVectors.remove(cnt);  
  
Toolkit.getDefaultToolkit().beep();  
        killCount++;  
        timer.setText(  
            ""  
            + (new Date().getTime()  
- baseTime)  
            + " / "  
            + killCount);
```

If the new separation is at least 25 units, the escape attempt is deemed successful and the prey fish will continue to live. *(You can decrease or increase the threshold distance used in the test in [Listing 21](#) to cause the prey fish object to be more or less successful in the escape attempt. If you make the threshold distance small enough, the prey fish will almost always escape.)*

When the escape attempt is not successful...

When the escape attempt is not successful, the remaining code in [Listing 21](#) is executed. The prey fish is deemed to have been eaten by the predator. Therefore, the prey fish and its direction vector are removed from the containers that contain them. This causes the prey fish to be removed from the population.

In addition, the program sounds a beep to notify the user of the successful attack by the predator and the kill count that is displayed in the bottom right of [Figure 1](#) is incremented by one.

Display elapsed time and number of fish eaten

Finally, the code in [Listing 21](#) displays the elapsed time in milliseconds since the **Start** button was clicked along with the number of fish that have been eaten. Note that the value that is displayed is the elapsed time when the most recent prey fish was eaten by the predator and is not the total elapsed time since the **Start** button was clicked. *(The elapsed time value won't change again until another prey fish is eaten.)*

When all of the prey fish have been eaten, the final time that is displayed is the time that was required for the predator to eat all of the fish in the population and the final kill count that is displayed is the same of the original number of prey fish in the population. *(Also note that the player must click the **Stop** button before starting a new game. Fixing this inconvenience will be a good exercise for the student.)*

When the escape attempt is successful...

When the escape attempt is successful, a clone of the prey fish in its new location is stored in the **preyObjects** container and a normalized version of the fish's direction vector is stored in the **displayVectors** container as shown in [Listing 22](#).

Listing 22 . Restore the prey fish to the population.

Listing 22 . Restore the prey fish to the population.

```
        }else{
            //The escape attempt was
successful. Restore
            // a clone of the prey object in
its new
            // location along with its
direction vector
            // to the population.

preyObjects.set(cnt,tempPrey.clone());
            //Save a normalized version of
the direction
            // vector for drawing later. It
will
            // overwrite the previously
saved vector and
            // if you watch closely it will
show the
            // prey object running away from
the
            // predator.
            displayVectors.set(

cnt,tempVectorA.normalize().scale(15.0));
        }//end else
    }//end if distance < 50
}//end for loop on population
```

[Listing 22](#) also signals the end of the **for** loop that began in [Listing 19](#).

Keep the prey fish in the playing field

I have one more section of code to explain that deals with the behavior of the prey fish during each iteration of the animation loop. The code in [Listing 23](#) causes prey fish that stray outside the playing field to return to the playing field.

Listing 23 . Keep the prey fish in the playing field.

```
        for(int cnt = 0;cnt <
preyObjects.size();cnt++){
            tempPrey = preyObjects.get(cnt);

if(tempPrey.getDisplacementVector(origin).
                                getLength() >
0.5*osiHeight){

            tempVectorA =

tempPrey.getDisplacementVector(origin);
            tempPrey =
tempPrey.addVectorToPoint(

tempVectorA.scale(0.1));

preyObjects.set(cnt,tempPrey.clone());
            }//end if prey object is out of
bounds
        }//end for loop to process each prey
object
```

The playing field is the interior of a sphere

The playing field is defined to be the interior of a sphere that is centered on the origin with a radius that is one-half the height of the off-screen image. The **for** loop in [Listing 23](#) iterates once for each prey fish remaining in the population. The **if** statement in [Listing 23](#) tests to determine if a prey fish is outside of the sphere.

If the prey fish is outside the sphere, the code in [Listing 23](#) gives that prey fish a nudge back toward the origin and saves a clone of the prey fish in its new location.

Erase the off-screen image and draw the large circle

[Listing 24](#) erases the off-screen image and draws the large circle that defines the playing field shown in [Figure 1](#).

Listing 24 . Erase the off-screen image and draw the large circle.

```
//Erase the screen
g2D.setColor(Color.WHITE);
GM01.fillRect(g2D, -
osiWidth/2,osiHeight/2,
osiWidth,osiHeight);

//Draw a broken-line circle that
represents the
// intersection of the spherical
boundary with the
// x-y plane. Although the prey objects
```

Listing 24 . Erase the off-screen image and draw the large circle.

and the

```
// predator can stray outside this
sphere, they
// prefer to be inside the sphere.
GM01.Point3D tempPointA = new
GM01.Point3D(
    new
GM01.ColMatrix3D(osiHeight/2,0,0));
    GM01.Point3D tempPointB;
    //The circle is defined by 39 points
around the
    // circumference.
    g2D.setColor(Color.BLACK);
    for(int cnt = 0;cnt < 39;cnt++){
        tempPointB =
            new GM01.Point3D(new
GM01.ColMatrix3D(
    osiHeight/2*Math.cos((cnt*360/39)*Math.PI/180),
    osiHeight/2*Math.sin((cnt*360/39)*Math.PI/180),
        0));

        //Connect every third pair of points
with a line
        if(cnt%3 == 0){
            new GM01.Line3D(
tempPointA,tempPointB).draw(g2D);
        }//end if
        //Save the old point.
        tempPointA = tempPointB;
    }//end for loop

    //Draw the final line required to close
```

Listing 24 . Erase the off-screen image and draw the large circle.

```
the circle
        new GM01.Line3D(tempPointA, new
GM01.Point3D(
                                new
GM01.ColMatrix3D(
osiHeight/2, 0, 0))) .draw(g2D);
```

Although somewhat tedious, the code in [Listing 24](#) is straightforward and shouldn't require an explanation beyond the embedded comments.

Draw the prey fish on the off-screen image

Depending on the states of the two check boxes in [Figure 1](#), [Listing 25](#) draws the **GM01.Point3D** objects that represent the prey fish or **GM01.Vector3D** objects that represent the direction vectors for the prey fish, or both, on the off-screen image. (*Note that only the vectors were drawn on the off-screen image in [Figure 1](#).*)

Listing 25 . Draw the prey fish on the off-screen image.

Listing 25 . Draw the prey fish on the off-screen image.

```
        g2D.setColor(Color.RED);
        for(int cnt = 0;cnt <
preyObjects.size();cnt++){
            tempPrey = preyObjects.get(cnt);
            if(drawPoints){
                tempPrey.draw(g2D); //draw circle
around point
            } //end if

            if(drawVectors){
                //Draw the vector with its tail at
the point.

displayVectors.get(cnt).draw(g2D, tempPrey);
            } //end if
        } //end for loop
```

There is nothing new in [Listing 25](#), so I won't bore you with a detailed explanation of the code.

It's time to deal with the predator

So far, we have been dealing exclusively with code that controls the behavior of the prey fish. The time has come to deal with the code that controls the behavior of the predator.

Cause the predator to slowly circle the cluster of prey fish

When the player clicks the **Attack** button, a **boolean** variable named **attack** is set to **true**, causing the predator to enter attack mode during the next iteration of the animation loop. However, when the value of this variable is **false**, the predator is not in attack mode and its behavior is to swim slowly

around the cluster of prey fish encouraging them to bunch up into a smaller and tighter cluster.

The code that accomplishes this circling behavior is shown in [Listing 26](#).

Listing 26 . Cause the predator to slowly circle the cluster of prey fish.

```
        //When the predator is not in attack
mode, cause
        // it to slowly circle the cluster of
prey
        // objects.
        if(!attack){
            //Get a displacement vector pointing
from the
            // predator to the preyCenter.
            predatorVec =

predator.getDisplacementVector(preycenter);

            //Create a vector that is rotated
relative to
            // predatorVec. Note how each
component in
            // this new vector is set equal to a
different
            // component in predatorVec
            tempVectorB = new GM01.Vector3D(
                new GM01.ColMatrix3D(
```

Listing 26 . Cause the predator to slowly circle the cluster of prey fish.

```
        predatorVec.getData(1),
        predatorVec.getData(2),

predatorVec.getData(0))).scale(0.15);

        //Scale predatorVec and add the two
vectors.
        // Then move the predator according
to the sum
        // of the vectors.
        //Moving the prey object in the
direction of
        // the sum of these two vectors
produces a
        // motion that causes the predator
to
        // spiral toward the preyCenter
instead of
        // simply moving in a straight line
toward the
        // preyCenter. The scale factors
control the
        // relative motion between the two
directions,
        // and are fairly sensitive.
        predatorVec =

predatorVec.scale(0.095).add(tempVectorB);

        predator =
predator.addVectorToPoint(
predatorVec.scale(1.0));
```

The code in [Listing 26](#) is very similar to code that I explained earlier in conjunction with the behavior of the prey fish. Therefore, no explanation beyond the embedded comments should be required.

Execute the attack

When the user clicks the **Attack** button, the value of the variable named **attack** is set to **true** , causing the code in [Listing 27](#) to be executed during subsequent iterations of the animation loop.

Listing 27 . Execute the attack.

Listing 27 . Execute the attack.

```
        }else{//attack is true
            predator =
predator.addVectorToPoint(
predatorVec.scale(0.25));

            //Check to see if the predator is
outside the
            // spherical boundary that defines
the playing
            // field.

if(predator.getDisplacementVector(origin).
                                getLength() >
0.5*osiHeight){
            //Shift out of attack mode and
start circling
            // the prey fish again.
            attack = false;
            attackButton.setEnabled(true);
        }//end if
    }//end else
```

The predator is in attack mode

The predator is in attack mode at the beginning of [Listing 27](#), and will remain in attack mode using the same displacement vector to control its speed and direction until it leaves the spherical playing field. When it leaves the playing field, the value of the **attack** variable will be set to **false** , causing the predator to revert to non-attack mode.

As the predator encounters prey fish along its trip toward the edge of the playing field, the code in [Listing 21](#) (*explained earlier*) will determine

whether those prey fish escape or get eaten by the predator.

Control of speed and direction

The displacement vector that controls the speed and direction of the predator (*predatorVec* in [Listing 27](#)) is created in the **actionPerformed** method in [Listing 3](#) in response to a click on the **Attack** button.

This vector is constructed to point to the most recently computed geometric center of the cluster of prey fish. Note however, that the vector may no longer point to the exact center of the cluster because the exact center of the cluster may have changed since it was last computed. In other words, the position of the geometric center of the prey-fish cluster changes as the predator attacks and causes the prey fish to scatter. As programmed, the predator is unable to respond to such changes and continues to move in the same direction at the same speed until it leaves the playing field.

In other words, even though the prey fish scatter, the predator is constrained to move in a straight line across the playing field once an attack has begun.

Note: An interesting upgrade:

A program upgrade to cause the predator to accommodate such changes in the geometric center would be an interesting exercise for the student.

Draw the predator on the off-screen image

[Listing 28](#) sets the drawing color to BLUE and draws the predator's direction vector on the off-screen image.

Listing 28 . Draw the predator on the off-screen image.

```
        g2D.setColor(Color.BLUE);
        //Enable the following statement to
draw a circle
        // around the point that represents
the predator.
        //predator.draw(g2D);

        //Draw the predator's vector.

predatorVec.normalize().scale(15.0).draw(
g2D, predator);
        g2D.setColor(Color.RED); //restore red
color
```

Copy off-screen image, insert time delay, etc

[Listing 29](#) copies the off-screen image to the canvas and then causes the animation thread to sleep for 166 milliseconds.

Listing 29 . Copy off-screen image, insert time delay, etc.

Listing 29 . Copy off-screen image, insert time delay, etc.

```
        //Copy the off-screen image to the
        canvas and then
        // do it all again.
        myCanvas.repaint();

        //Insert a time delay. Change the
        sleep time to
        // speed up or slow down the
        animation.
        try{
            Thread.currentThread().sleep(166);
        }catch(Exception e){
            e.printStackTrace();
        }//end catch
    }//end animation loop
} //end run method
} //end inner class named Animate
```

[Listing 29](#) also signals the end of the animation loop, the end of the **run** method, and the end of the inner class named **Animate** .

That is all of the code that I will explain for this program. You can view the remainder of the code in [Listing 31](#) near the end of the module.

Not a graphics program

Even though this program produces quite a lot of 3D graphics, and those graphics are important in the playing of the game, this is not a graphics program. Rather, it is a 3D data-processing program that produces graphics as a side effect.

That is not to say that the graphics are unimportant. To the contrary, the graphics provide visual feedback to the player, which allows the player to

implement a strategy for success. Without graphics, the game would be very boring. However, the programming effort to produce the graphics represents an almost trivial part of the total programming effort for this game.

An almost trivial part of the programming effort

Exclusive of the code required to draw the large circle shown in [Figure 1](#), all of the graphics shown in [Figure 1](#), [Figure 5](#), [Figure 6](#), [Figure 7](#), and [Figure 8](#) are produced by only two calls to game-math library methods named **draw** . In other words, all of the required graphics are produced by only two statements in the program code. *(One additional statement is required if you want to display the small circles that represent the locations of the prey fish.)*

If those two statements are removed, the program will still compile and run, and the game can still be played. However, without visual feedback, the game isn't much fun. The lack of visual feedback eliminates the strategy aspect of the game, causing it to be solely a game of chance. Be that as it may, with or without visual feedback, the player can still click the **Start** button and then repetitively click the **Attack** button until the display in the bottom-right of [Figure 1](#) shows that all of the prey fish have been eaten.

Why am I telling you this?

I'm telling you this to emphasize that the essential skills required to program this game (*and probably most games for that matter*) consist of skills in mathematics, programming logic, and several other technical areas. The ability to produce on-screen graphics is necessary for an enjoyable game, but, (*given a good game-math library that supports graphics*) , producing on-screen graphics is almost a trivial part of the programming effort. In this collection of modules, you need to be mainly concentrating on learning mathematics and programming logic and treating the production of on-screen graphics almost as an afterthought.

The program named GM01test04

This animation program is designed to exercise many of the 2D features of the **GM01** game-math library. The animation is generally based on the idea of a flocking behavior similar to that exhibited by birds and fish. A set of **GM01.Point2D** objects is created with random locations to act as predators. An additional **GM01.Point2D** object is also created to play the part of a prey object.

The prey object is drawn in red while the predators are drawn in black as shown in [Figure 2](#). An algorithm is executed that attempts to cause the predators to chase the prey object without colliding with one another.

Even though the algorithm causes the predators to chase the prey object, it also tries to keep the predators from colliding with the prey object.

The user input GUI

A GUI is provided that contains an input text field for the number of predators plus a **Start** button and a **Stop** button. The GUI also contains check boxes that allow the user to elect to display points only, direction vectors only, or both. *(Both the point and the direction vector is always displayed for the prey object.)*

The user specifies the number of randomly-placed predators and clicks the **Start** button, at which time the animation begins and the predators start chasing the prey object. Prey-object motion is random.

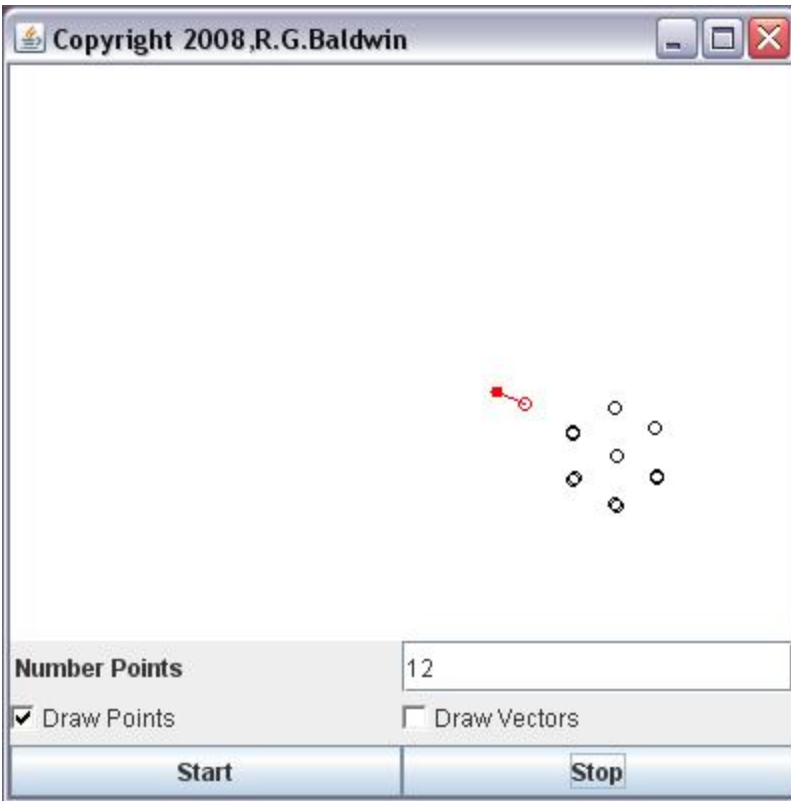
The animation continues until the user clicks the **Stop** button. The user can click the **Stop** button, change any of the input parameters, and then click the **Start** button again to re-start the animation with different parameters such as the number of predator objects.

Swimming in formation

An unexpected result is that the algorithm seems to cause the predators to come together and swim in formation while chasing the prey object. The most common formation is hexagonal as shown in [Figure 9](#), which shows 12 predators swimming in a hexagonal formation.

(Note that some of the twelve predators are hidden by other predators.)

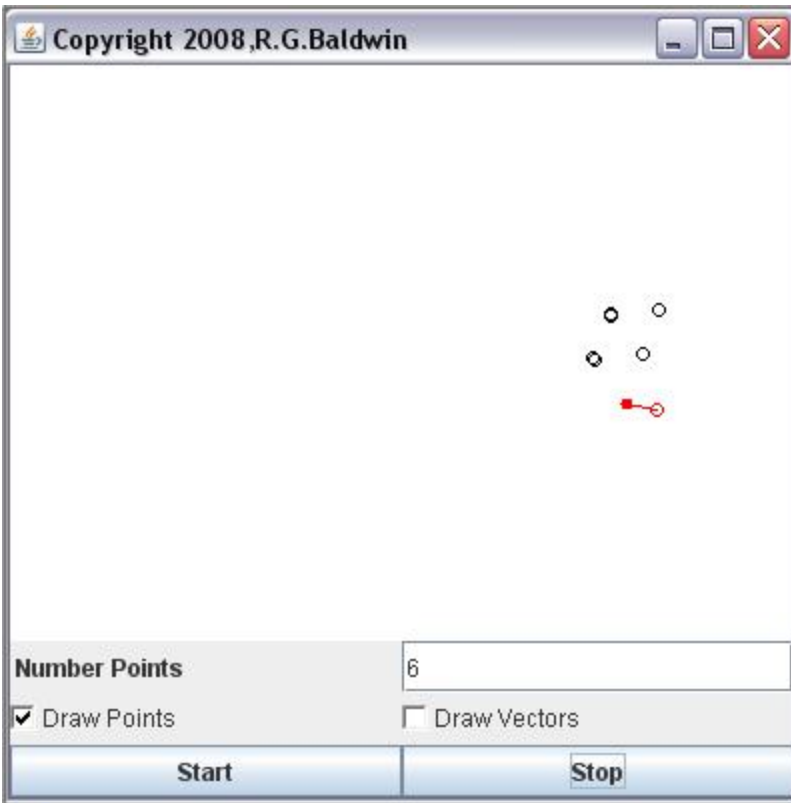
Figure 9 Twelve predators swimming in a hexagon formation in GM01test04.



Other formations appear as well

Some triangles, diamonds, and incomplete hexagons also appear. For example, [Figure 10](#) shows six predators swimming in a diamond formation.

Figure 10 Six predators swimming in a diamond formation.



No explanation for this behavior

I haven't given the matter a lot of thought, but at this point, I have no explanation for this behavior. Note that the tendency to swim in formation is more visually obvious when only the points are displayed. When the vectors are displayed, it is more difficult to pick out the formation.

Dogged determination

On the other hand, the animation is most impressive when the direction vectors are displayed, with or without points, because the vectors illustrate the dogged determination and undying focus that the predators maintain while chasing the prey object.

Won't explain the code

Once you understand the code in the program named **GM01test08** that I explained earlier in this module, you should have no difficulty understanding the code in this program. Therefore, I won't explain the code

in this program. I included this program in this module mainly to illustrate the differences between 2D and 3D from both a visual and programming viewpoint.

A complete listing of this program is provided in [Listing 32](#) near the end of the module.

The program named GM01test03

This is a 3D update of the 2D program named **GM01test04** discussed above.

A comparison of programming requirements

Basically all that was required to perform the update was to specify 3D classes from the game-math library in place of the 2D classes used in the 2D version of the program. In some cases, this in turn required that argument lists for constructors and methods be expanded from two dimensions to three dimensions. Just about everything else took care of itself.

A comparison of these two programs illustrates the value of the game-math library named **GM01** and the ease with which you can switch back and forth between 2D and 3D programming when using the library.

A comparison of visual behavior

The visual behavior of this 3D version, as shown in [Figure 3](#), is more realistic than the 2D version. This is particularly true when the prey object gets in the middle of the predators and the display is showing vectors. In the 2D version, a predator is constrained to swing around only in the plane of the screen. However, in this 3D version, a predator is not subject to that constraint and is free to swing around in the most appropriate way as the prey object passes by.

This constraint causes the motion in the 2D version to be less fluid than the motion in the 3D version. This can best be demonstrated with only one

predator because that makes it easy to see the behavior of an individual predator as the animation is running.

No swimming in formation

One very interesting thing that I have noticed is that unlike the 2D version, the predators in this 3D version don't seem to have a tendency to form up and swim in formation while chasing the prey object. This may be because they have more options in terms of avoiding collisions while giving chase. However, that is pure speculation on my part since I don't know why the predators tend to swim in formation in the 2D version anyway. *(It is also possible that the predators form into a 3D formation, which isn't visually obvious in the 2D projection.)*

Won't explain the code

As is the case with the earlier program named **GM01test04** , once you understand the code in the program named **GM01test08** , you should have no difficulty understanding the code in this program. Therefore, I won't explain the code in this program. I included this program and the earlier 2D version in this module mainly to illustrate the differences between 2D and 3D from both a visual viewpoint and programming viewpoint.

A complete listing of this program is provided in [Listing 33](#) near the end of the module.

The program namedStringArt04

This program animates the behavior of the earlier program named **StringArt03** that I explained in an earlier module. See the comments at the beginning of that program for a description of both programs.

The only significant difference in the behavior of the two programs is that this program slows the rotation process down and animates it so that the user can see it happening in slow motion. Of course, quite a few changes

were required to convert the program from a static program to an animated program.

However, if you understand the code in the earlier program named **StringArt03** and you understand the code in the program named **GM01test08** that I explained earlier in this module, you should have no difficulty understanding the code in the program named **StringArt04** . Therefore, I won't explain the code in this program. A screen shot of the program in action is shown in [Figure 4](#) . A complete listing of the program is provided in [Listing 34](#) .

Visual output

When viewing the output, remember that the program executes the rotations around the axes sequentially in the following order:

- z-axis
- x-axis
- y-axis

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter6 - Scaling and Unit Vectors* .

The homework assignment for the next module is to study the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* .

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 30](#) through [Listing 34](#). Compile the code and execute it in conjunction with the game-math library named **GM01** provided in [Listing 30](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to write your first interactive 3D game using the game-math library named **GM01**. You also learned how to write a Java program that simulates flocking behavior such as that exhibited by birds and fish and you learned how to incorporate that behavior into the game. Finally, you examined three other programs that illustrate various aspects of both 2D and 3D animation using the game-math library.

What's next?

In the next module, you will learn the fundamentals of the *vector dot product* in both 2D and 3D. You will learn how to update the game-math library to support various aspects of the vector dot product, and you will learn how to write 2D and 3D programs that use the vector dot product methods in the game-math library.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0140: Our First 3D Game Program
- File: Game0140.htm
- Published: 10/20/12

- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 30](#) through [Listing 34](#) below.

Listing 30 . Source code for the game-math library named GM01.

```
/*GM01.java  
Copyright 2008, R.G.Baldwin  
Revised 02/24/08
```

This is a major upgrade to the game-math library.
This

version upgrades the version named GM004 to a new

version upgrades the version named GM2D04 to a new version named simply GM01.

The primary purpose of the upgrade was to add 3D capability for all of the 2D features provided by the previous version. Because both 2D and 3D capabilities are included, it is no longer necessary to differentiate between the two in the name of the class. Therefore, this version is named GM01.

Adding 3D capability entailed major complexity in one particular area: drawing the objects. It is difficult to draw a 3D object on a 2D screen. This requires a projection process to project each point in the 3D object onto the correct location on a 2D plane. There are a variety of ways to do this. This 3D library uses an approach often referred to as an oblique parallel projection. See the following URL for technical information on the projection process:

<http://local.wasp.uwa.edu.au/~pbourke/geometry/classification/>

In addition to adding 3D capability, this version also eliminates the confusion surrounding the fact that the

default direction of the positive y-axis is going down the screen instead of up the screen as viewers have become accustomed to. When you use this library, you can program under the assumption that the positive direction of the y-axis is up the screen, provided you funnel all of your drawing tasks through the library and don't draw directly on the screen.

The name GMnn is an abbreviation for GameMathnn.

See the file named GM2D01.java for a general description of the game-math library. The library has been updated several times. This file is an update of GM2D04.

In addition to the updates mentioned above, this update cleaned up some lingering areas of code inefficiency, using the simplest available method to draw on an off-screen image. In addition, the following new methods were added:

The following methods are new static methods of the class named GM01. The first method in the list deals with the problem of displaying a 3D image on a 3D screen.

The last five methods in the list wrap the

standard
graphics methods for the purpose of eliminating
the issue
of the direction of the positive Y-axis.

GM01.convert3Dto2D
GM01.translate
GM01.drawLine
GM01.fillOval
GM01.drawOval
GM01.fillRect

The following methods are new instance methods of
the
indicated static top-level classes belonging to
the class
named GM01.

GM01.Vector2D.scale
GM01.Vector2D.negate
GM01.Point2D.clone
GM01.Vector2D.normalize
GM01.Point2D.rotate
GM01.Point2D.scale

GM01.Vector3D.scale
GM01.Vector3D.negate
GM01.Point3D.clone
GM01.Vector3D.normalize
GM01.Point3D.rotate
GM01.Point3D.scale

Tested using JDK 1.6 under WinXP.

```
*****  
*****/  
import java.awt.geom.*;  
import java.awt.*;
```

```

public class GM01{
    //-----
    -----//

    //This method converts a ColMatrix3D object to a
    // ColMatrix2D object. The purpose is to accept
    // x, y, and z coordinate values and transform
those
    // values into a pair of coordinate values
suitable for
    // display in two dimensions.
    //See
http://local.wasp.uwa.edu.au/~pbourke/geometry/
    // classification/ for technical background on
the
    // transform from 3D to 2D.
    //The transform equations are:
    //  $x_{2d} = x_{3d} + z_{3d} * \cos(\theta) / \tan(\alpha)$ 
    //  $y_{2d} = y_{3d} + z_{3d} * \sin(\theta) / \tan(\alpha)$ ;
    //Let  $\theta = 30$  degrees and  $\alpha = 45$  degrees
    //Then: $\cos(\theta) = 0.866$ 
    //       $\sin(\theta) = 0.5$ 
    //       $\tan(\alpha) = 1$ ;
    //Note that the signs in the above equations
depend
    // on the assumed directions of the angles as
well as
    // the assumed positive directions of the axes.
The

    // signs used in this method assume the
following:
    //      Positive x is to the right.
    //      Positive y is up the screen.
    //      Positive z is protruding out the front of
the
    //      screen.

```

```

    //    The viewing position is above the x axis
and to the
    //    right of the z-y plane.
    public static GM01.ColMatrix2D convert3Dto2D(

GM01.ColMatrix3D data){
    return new GM01.ColMatrix2D(
        data.getData(0) -
0.866*data.getData(2),
        data.getData(1) -
0.50*data.getData(2));
    }//end convert3Dto2D
    //-----
-----//

    //This method wraps around the translate method
of the
    // Graphics2D class. The purpose is to cause the
    // positive direction for the y-axis to be up
the screen
    // instead of down the screen. When you use this
method,
    // you should program as though the positive
direction
    // for the y-axis is up.
    public static void translate(Graphics2D g2D,
                                double xOffset,
                                double yOffset){
        //Flip the sign on the y-coordinate to change
the
        // direction of the positive y-axis to go up
the
        // screen.
        g2D.translate(xOffset, -yOffset);
    }//end translate
    //-----
-----//

```

```

    //This method wraps around the drawLine method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void drawLine(Graphics2D g2D,
                                double x1,
                                double y1,
                                double x2,
                                double y2){
        //Flip the sign on the y-coordinate value.
        g2D.drawLine((int)x1, -(int)y1, (int)x2, -
(int)y2);
    }//end drawLine
    //-----
-----//

```

```

    //This method wraps around the fillOval method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void fillOval(Graphics2D g2D,
                                double x,

```

```

                                double y,
                                double width,
                                double height){
    //Flip the sign on the y-coordinate value.
    g2D.fillOval((int)x, -(int)y, (int)width,
(int)height);
    }//end fillOval
    //-----
-----//

```

```

    //This method wraps around the drawOval method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for

```

```

    // the y-axis is up.
    public static void drawOval(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
    //Flip the sign on the y-coordinate value.
    g2D.drawOval((int)x, -(int)y, (int)width,
(int)height);
    }//end drawOval
    //-----
-----//

```

```

    //This method wraps around the fillRect method
of the
    // Graphics class. The purpose is to cause the
positive

```

```

    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void fillRect(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.fillRect((int)x, -(int)y, (int)width,
(int)height);
    }//end fillRect
    //-----
-----//

```

```

    //An object of this class represents a 2D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix2D{
        double[] data = new double[2];

        public ColMatrix2D(double data0, double data1){
            data[0] = data0;
            data[1] = data1;
        }//end constructor
        //-----
-----//

```



```

//Overridden toString method.
public String toString(){
    return data[0] + "," + data[1];
}//end overridden toString method
//-----
-----//

public double getData(int index){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        return data[index];
    }//end else
}//end getData method
//-----
-----//

public void setData(int index,double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        this.data[index] = data;
    }//end else
}//end setData method
//-----
-----//

//This method overrides the equals method
inherited
// from the class named Object. It compares
the values
// stored in two matrices and returns true if
the
// values are equal or almost equal and
returns false
// otherwise.
public boolean equals(Object obi){

```

```

        if(obj instanceof GM01.ColMatrix2D &&
Math.abs(((GM01.ColMatrix2D)obj).getData(0) -
                                                getData(0)) <=
0.00001 &&
Math.abs(((GM01.ColMatrix2D)obj).getData(1) -
                                                getData(1)) <=
0.00001){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
    -----//

    //Adds one ColMatrix2D object to another
    ColMatrix2D
    // object, returning a ColMatrix2D object.
    public GM01.ColMatrix2D add(GM01.ColMatrix2D
matrix){
        return new GM01.ColMatrix2D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1));
    }//end add
    //-----
    -----//

    //Subtracts one ColMatrix2D object from
    another
    // ColMatrix2D object, returning a ColMatrix2D
    object.
    // The object that is received as an incoming

```

```

        // parameter is subtracted from the object on
which
        // the method is called.
        public GM01.ColMatrix2D subtract(
                                GM01.ColMatrix2D
matrix){
        return new GM01.ColMatrix2D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1));
        }//end subtract
        //-----
-----//
    }//end class ColMatrix2D

//=====
====//

```

```

    //An object of this class represents a 3D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix3D{
        double[] data = new double[3];

        public ColMatrix3D(
                                double data0,double data1,double
data2){
            data[0] = data0;
            data[1] = data1;
            data[2] = data2;
        }//end constructor
        //-----

```

```

-----//

    public String toString(){
        return data[0] + "," + data[1] + "," +
data[2];
    }//end overridden toString method
    //-----
-----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return data[index];
        }//end else
    }//end getData method
    //-----
-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            this.data[index] = data;
        }//end else
    }//end setData method
    //-----
-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in two matrices and returns true if
the
    // values are equal or almost equal and
returns false

```

```

        // otherwise.
        public boolean equals(Object obj){
            if(obj instanceof GM01.ColMatrix3D &&
Math.abs(((GM01.ColMatrix3D)obj).getData(0) -
                                                getData(0)) <=
0.00001 &&
Math.abs(((GM01.ColMatrix3D)obj).getData(1) -
                                                getData(1)) <=
0.00001 &&
Math.abs(((GM01.ColMatrix3D)obj).getData(2) -
                                                getData(2)) <=
0.00001){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
        -----//

        //Adds one ColMatrix3D object to another
        ColMatrix3D
        // object, returning a ColMatrix3D object.
        public GM01.ColMatrix3D add(GM01.ColMatrix3D
matrix){
            return new GM01.ColMatrix3D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1),

getData(2)+matrix.getData(2));
        }//end add

```

```

//-----
-----//

//Subtracts one ColMatrix3D object from
another
// ColMatrix3D object, returning a ColMatrix3D
object.
// The object that is received as an incoming
// parameter is subtracted from the object on
which
// the method is called.
public GM01.ColMatrix3D subtract(
                                GM01.ColMatrix3D
matrix){
    return new GM01.ColMatrix3D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1),
                                getData(2)-
matrix.getData(2));
    }//end subtract
//-----
-----//
} //end class ColMatrix3D

//=====
====//

//=====
====//

public static class Point2D{
    GM01.ColMatrix2D point;

    public Point2D(GM01.ColMatrix2D point)

```

```

{ //constructor
    //Create and save a clone of the ColMatrix2D
object
    // used to define the point to prevent the
point
    // from being corrupted by a later change in
the
    // values stored in the original ColMatrix2D
object
    // through use of its set method.
    this.point = new ColMatrix2D(
point.getData(0),point.getData(1));
} //end constructor
//-----
-----//

    public String toString(){
        return point.getData(0) + "," +
point.getData(1);
    } //end toString
    //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        } else{
            return point.getData(index);
        } //end else
    } //end getData
    //-----
    -----//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();

```

```

        }else{
            point.setData(index,data);
        }//end else
    }//end setData
    //-----
-----//

    //This method draws a small circle around the
location
    // of the point on the specified graphics
context.
    public void draw(Graphics2D g2D){
        drawOval(g2D,getData(0)-3,
                getData(1)+3,6,6);
    }//end draw

    //-----
-----//

    //Returns a reference to the ColMatrix2D
object that
    // defines this Point2D object.
    public GM01.ColMatrix2D getColMatrix(){
        return point;
    }//end getColMatrix
    //-----
-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix2D objects that
define two
    // Point2D objects and returns true if they
are equal
    // and false otherwise.

```



```

        public boolean equals(Object obj){
            if(point.equals(((GM01.Point2D)obj).
getColMatrix())){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
        -----//

        //Gets a displacement vector from one Point2D
object
        // to a second Point2D object. The vector
points from
        // the object on which the method is called to
the
        // object passed as a parameter to the method.
Kjell
        // describes this as the distance you would
have to
        // walk along the x and then the y axes to get
from
        // the first point to the second point.
        public GM01.Vector2D getDisplacementVector(
                                GM01.Point2D
point){
            return new GM01.Vector2D(new
GM01.ColMatrix2D(
                                point.getData(0)-
getData(0),
                                point.getData(1)-
getData(1)));
        }//end getDisplacementVector
        //-----

```

```

-----//

    //Adds a Vector2D to a Point2D producing a
    // new Point2D.
    public GM01.Point2D addVectorToPoint(
GM01.Vector2D vec){
        return new GM01.Point2D(new
GM01.ColMatrix2D(
                                getData(0) +
vec.getData(0),
                                getData(1) +
vec.getData(1)));
    }//end addVectorToPoint
    //-----
-----//

    //Returns a new Point2D object that is a clone
of
    // the object on which the method is called.
    public Point2D clone(){
        return new Point2D(
                                new
ColMatrix2D(getData(0),getData(1)));
    }//end clone
    //-----
-----//

    //The purpose of this method is to rotate a
point
    // around a specified anchor point in the x-y
plane.
    //The rotation angle is passed in as a double
value
    // in degrees with the positive angle of
rotation
    // being counter-clockwise.

```

```

    //This method does not modify the contents of
the
    // Point2D object on which the method is
called.
    // Rather, it uses the contents of that object
to
    // instantiate, rotate, and return a new
Point2D
    // object.
    //For simplicity, this method translates the
    // anchorPoint to the origin, rotates around
the
    // origin, and then translates back to the
    // anchorPoint.
    /*
    See
http://www.ia.hiof.no/~borres/cgraph/math/threed/
    p-threed.html for a definition of the
equations
    required to do the rotation.

    x2 = x1*cos - y1*sin
    y2 = x1*sin + y1*cos
    */
    public GM01.Point2D rotate(GM01.Point2D
anchorPoint,
                                double angle){
        GM01.Point2D newPoint = this.clone();

        double tempX ;
        double tempY;

        //Translate anchorPoint to the origin
        GM01.Vector2D tempVec =
            new
GM01.Vector2D(anchorPoint.getColMatrix());
        newPoint =

```

```

newPoint.addVectorToPoint(tempVec.negate());

    //Rotate around the origin.
    tempX = newPoint.getData(0);
    tempY = newPoint.getData(1);
    newPoint.setData(new x coordinate
                      0,

tempX*Math.cos(angle*Math.PI/180) -
tempY*Math.sin(angle*Math.PI/180));

    newPoint.setData(new y coordinate
                      1,

tempX*Math.sin(angle*Math.PI/180) +
tempY*Math.cos(angle*Math.PI/180));

    //Translate back to anchorPoint
    newPoint =
newPoint.addVectorToPoint(tempVec);

    return newPoint;

} //end rotate
//-----
-----//

    //Multiplies this point by a scaling matrix
received
    // as an incoming parameter and returns the
scaled
    // point.
    public GM01.Point2D scale(GM01.ColMatrix2D
scale){

```

```

        return new GM01.Point2D(new ColMatrix2D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1)));
    }//end scale
    //-----
-----//
} //end class Point2D

//=====
====//

    public static class Point3D{
        GM01.ColMatrix3D point;

        public Point3D(GM01.ColMatrix3D point)
    { //constructor
        //Create and save a clone of the ColMatrix3D
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix3D
object
        // through use of its set method.
        this.point =
            new ColMatrix3D(point.getData(0),
                                point.getData(1),
                                point.getData(2));
    } //end constructor
    //-----
-----//

    public String toString(){

```

```

        return point.getData(0) + "," +
point.getData(1)
                                + "," +
point.getData(2);
    }//end toString
    //-----
-----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return point.getData(index);
        }//end else
    }//end getData
    //-----
-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            point.setData(index,data);
        }//end else
    }//end setData
    //-----
-----//

    //This method draws a small circle around the
location

    // of the point on the specified graphics
context.
    public void draw(Graphics2D g2D){

        //Get 2D projection coordinate values.
        ColMatrix2D temp = convert3Dto2D(point);
        drawOval(g2D,temp.getData(0)-3,

```

```

        temp.getData(1)+3,
        6,
        6);
    }//end draw
    //-----
-----//

    //Returns a reference to the ColMatrix3D
object that
    // defines this Point3D object.
    public GM01.ColMatrix3D getColMatrix(){
        return point;
    }//end getColMatrix
    //-----
-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix3D objects that
define two
    // Point3D objects and returns true if they
are equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(point.equals(((GM01.Point3D)obj).
getColMatrix())){
            return true;

        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
-----//

```

```

        //Gets a displacement vector from one Point3D
object
        // to a second Point3D object. The vector
points from
        // the object on which the method is called to
the
        // object passed as a parameter to the method.
Kjell
        // describes this as the distance you would
have to
        // walk along the x and then the y axes to get
from
        // the first point to the second point.
        public GM01.Vector3D getDisplacementVector(
                                                GM01.Point3D
point){
            return new GM01.Vector3D(new
GM01.ColMatrix3D(
                                point.getData(0)-
getData(0),
                                point.getData(1)-
getData(1),
                                point.getData(2)-
getData(2)));
        }//end getDisplacementVector
        //-----
        -----//

        //Adds a Vector3D to a Point3D producing a
        // new Point3D.
        public GM01.Point3D addVectorToPoint(
GM01.Vector3D vec){
            return new GM01.Point3D(new
GM01.ColMatrix3D(
                                getData(0) +

```



```

vec.getData(0),
vec.getData(1),
vec.getData(2));
    }//end addVectorToPoint
    //-----
    -----//

```

```

    //Returns a new Point3D object that is a clone
of
    // the object on which the method is called.
    public Point3D clone(){
        return new Point3D(new
ColMatrix3D(getData(0),
getData(1),
getData(2)));
    }//end clone
    //-----
    -----//

```

```

    //The purpose of this method is to rotate a
point
    // around a specified anchor point in the
following
    // order:
    // Rotate around z - rotation in x-y plane.
    // Rotate around x - rotation in y-z plane.
    // Rotate around y - rotation in x-z plane.
    //The rotation angles are passed in as double
values
    // in degrees (based on the right-hand rule)
in the
    // order given above, packaged in an object of
the

```

```

    // class GM01.ColMatrix3D. (Note that in this
case,
    // the ColMatrix3D object is simply a
convenient
    // container and it has no significance from a
matrix
    // viewpoint.)
    //The right-hand rule states that if you point
the
    // thumb of your right hand in the positive
direction
    // of an axis, the direction of positive
rotation
    // around that axis is given by the direction
that
    // your fingers will be pointing.
    //This method does not modify the contents of
the
    // Point3D object on which the method is
called.
    // Rather, it uses the contents of that object
to
    // instantiate, rotate, and return a new
Point3D
    // object.
    //For simplicity, this method translates the
    // anchorPoint to the origin, rotates around
the
    // origin, and then translates back to the
    // anchorPoint.

/*
See
http://www.ia.hiof.no/~borres/cgraph/math/threed/
    p-threed.html for a definition of the
equations
    required to do the rotation.
z-axis

```

```
x2 = x1*cos - y1*sin  
y2 = x1*sin + y1*cos
```

x-axis

```
y2 = y1*cos(v) - z1*sin(v)  
z2 = y1*sin(v) + z1*cos(v)
```

y-axis

```
x2 = x1*cos(v) + z1*sin(v)  
z2 = -x1*sin(v) + z1*cos(v)  
*/
```

```
public GM01.Point3D rotate(GM01.Point3D  
anchorPoint,  
                                GM01.ColMatrix3D  
angles){  
    GM01.Point3D newPoint = this.clone();  
  
    double tempX ;  
    double tempY;  
    double tempZ;  
  
    //Translate anchorPoint to the origin  
    GM01.Vector3D tempVec =  
        new  
GM01.Vector3D(anchorPoint.getColMatrix());  
    newPoint =  
  
newPoint.addVectorToPoint(tempVec.negate());  
  
    double zAngle = angles.getData(0);  
    double xAngle = angles.getData(1);  
    double yAngle = angles.getData(2);  
  
    //Rotate around z-axis  
    tempX = newPoint.getData(0);  
    tempY = newPoint.getData(1);  
    newPoint.setData(new x coordinate
```

0,

tempX*Math.cos(zAngle*Math.PI/180) -

tempY*Math.sin(zAngle*Math.PI/180));

newPoint.setData("//new y coordinate
1,

tempX*Math.sin(zAngle*Math.PI/180) +

tempY*Math.cos(zAngle*Math.PI/180));

//Rotate around x-axis
tempY = newPoint.getData(1);
tempZ = newPoint.getData(2);
newPoint.setData("//new y coordinate
1,

tempY*Math.cos(xAngle*Math.PI/180) -

tempZ*Math.sin(xAngle*Math.PI/180));

newPoint.setData("//new z coordinate
2,

tempY*Math.sin(xAngle*Math.PI/180) +

tempZ*Math.cos(xAngle*Math.PI/180));

//Rotate around y-axis
tempX = newPoint.getData(0);
tempZ = newPoint.getData(2);
newPoint.setData("//new x coordinate
0,

tempX*Math.cos(yAngle*Math.PI/180) +

```

tempZ*Math.sin(yAngle*Math.PI/180));

        newPoint.setData(new z coordinate
                        2,
                        -
tempX*Math.sin(yAngle*Math.PI/180) +
tempZ*Math.cos(yAngle*Math.PI/180));

        //Translate back to anchorPoint
        newPoint =
newPoint.addVectorToPoint(tempVec);

        return newPoint;

    }//end rotate
    //-----
    -----//

    //Multiplies this point by a scaling matrix
received
    // as an incoming parameter and returns the
scaled
    // point.
    public GM01.Point3D scale(GM01.ColMatrix3D
scale){
        return new GM01.Point3D(new ColMatrix3D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1),
                                getData(2) *
scale.getData(2)));
    }//end scale
    //-----
    -----//

```

```

    }//end class Point3D

//=====
====//

//=====
====//

    public static class Vector2D{
        GM01.ColMatrix2D vector;

        public Vector2D(GM01.ColMatrix2D vector)
        {//constructor
            //Create and save a clone of the ColMatrix2D
            object
            // used to define the vector to prevent the
            vector
            // from being corrupted by a later change in
            the
            // values stored in the original ColVector2D
            object.
            this.vector = new ColMatrix2D(
vector.getData(0),vector.getData(1));
            }//end constructor
            //-----
            -----//

            public String toString(){
                return vector.getData(0) + "," +
vector.getData(1);
            }//end toString
            //-----
            -----//

            public double getData(int index){

```

```

        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
    //-----
-----//

```

```

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData
    //-----
-----//

```

```

    //This method draws a vector on the specified
graphics
    // context, with the tail of the vector
located at a
    // specified point, and with a small filled
circle at
    // the head.
    public void draw(Graphics2D g2D,GM01.Point2D
tail){

        drawLine(g2D,

                    tail.getData(0),
                    tail.getData(1),
                    tail.getData(0)+vector.getData(0),
                    tail.getData(1)+vector.getData(1));

        fillOval(g2D,

```

```

tail.getData(0)+vector.getData(0)-3,
tail.getData(1)+vector.getData(1)+3,
        6,
        6);
    }//end draw
    //-----
-----//

    //Returns a reference to the ColMatrix2D
object that
    // defines this Vector2D object.
    public GM01.ColMatrix2D getColMatrix(){
        return vector;
    }//end getColMatrix
    //-----
-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix2D objects that
define two
    // Vector2D objects and returns true if they
are equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(vector.equals((
(GM01.Vector2D)obj).getColMatrix())){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method

```



```
        //-----  
-----//
```

```
        //Adds this vector to a vector received as an  
incoming
```

```
        // parameter and returns the sum as a vector.  
        public GM01.Vector2D add(GM01.Vector2D vec){  
            return new GM01.Vector2D(new ColMatrix2D(  
vec.getData(0)+vector.getData(0),  
vec.getData(1)+vector.getData(1)));  
        }//end add
```

```
        //-----  
-----//
```

```
        //Returns the length of a Vector2D object.
```

```
        public double getLength(){  
            return Math.sqrt(  
                getData(0)*getData(0) +  
getData(1)*getData(1));  
        }//end getLength
```

```
        //-----  
-----//
```

```
        //Multiplies this vector by a scale factor  
received as
```

```
        // an incoming parameter and returns the  
scaled  
        // vector.
```

```
        public GM01.Vector2D scale(Double factor){  
            return new GM01.Vector2D(new ColMatrix2D(  
                getData(0) *  
factor,  
                getData(1) *  
factor));  
        }//end scale
```

```

        //-----
        -----//

        //Changes the sign on each of the vector
        components
        // and returns the negated vector.
        public GM01.Vector2D negate(){
            return new GM01.Vector2D(new ColMatrix2D(
                                                                -
getData(0),
                                                                -
getData(1)));
        }//end negate
        //-----
        -----//

        //Returns a new vector that points in the same
        // direction but has a length of one unit.
        public GM01.Vector2D normalize(){
            double length = getLength();
            return new GM01.Vector2D(new ColMatrix2D(

getData(0)/length,

getData(1)/length));
        }//end normalize
        //-----
        -----//
    }//end class Vector2D

//=====
====//

    public static class Vector3D{
        GM01.ColMatrix3D vector;
    }

```

```

        public Vector3D(GM01.ColMatrix3D vector)
{
    //constructor
        //Create and save a clone of the ColMatrix3D
object
        // used to define the vector to prevent the
vector
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix3D
object.
        this.vector = new
ColMatrix3D(vector.getData(0),
vector.getData(1),
vector.getData(2));
    }//end constructor
    //-----
    -----//

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1)
                                + "," +
vector.getData(2);
    }//end toString
    //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
    //-----

```

-----//

```
public void setData(int index,double data){
    if((index < 0) || (index > 2)){
        throw new IndexOutOfBoundsException();
    }else{
        vector.setData(index,data);
    }//end else
}//end setData
//-----
```

-----//

```
//This method draws a vector on the specified
graphics
// context, with the tail of the vector
located at a
// specified point, and with a small circle at
the
// head.
public void draw(Graphics2D g2D,GM01.Point3D
tail){
```

```
    //Get a 2D projection of the tail
    GM01.ColMatrix2D tail2D =
convert3Dto2D(tail.point);
```

```
    //Get the 3D location of the head
    GM01.ColMatrix3D head =

tail.point.add(this.getColMatrix());
```

```
    //Get a 2D projection of the head
    GM01.ColMatrix2D head2D =
convert3Dto2D(head);
    drawLine(g2D,tail2D.getData(0),
                tail2D.getData(1),
                head2D.getData(0),
```

```

        head2D.getData(1));

    //Draw a small filled circle to identify the
head.
        fillOval(g2D,head2D.getData(0)-3,
                head2D.getData(1)+3,
                6,
                6);

    }//end draw
    //-----
    -----//

    //Returns a reference to the ColMatrix3D
object that
    // defines this Vector3D object.
    public GM01.ColMatrix3D getColMatrix(){
        return vector;
    }//end getColMatrix
    //-----
    -----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix3D objects that
define two
    // Vector3D objects and returns true if they
are equal

    // and false otherwise.
    public boolean equals(Object obj){
        if(vector.equals((
(GM01.Vector3D)obj).getColMatrix())){
            return true;
        }else{

```

```

        return false;
    }//end else

    }//end overridden equals method
    //-----
    -----//

    //Adds this vector to a vector received as an
incoming
    // parameter and returns the sum as a vector.
    public GM01.Vector3D add(GM01.Vector3D vec){
        return new GM01.Vector3D(new ColMatrix3D(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1),
vec.getData(2)+vector.getData(2)));
    }//end add
    //-----
    -----//

    //Returns the length of a Vector3D object.
    public double getLength(){
        return Math.sqrt(getData(0)*getData(0) +
                           getData(1)*getData(1) +
                           getData(2)*getData(2));
    }//end getLength
    //-----
    -----//

    //Multiplies this vector by a scale factor
received as
    // an incoming parameter and returns the
scaled
    // vector.
    public GM01.Vector3D scale(Double factor){

```

```

        return new GM01.Vector3D(new ColMatrix3D(
            getData(0) *
factor,
            getData(1) *
factor,
            getData(2) *
factor));
    } //end scale
    //-----
    -----//

    //Changes the sign on each of the vector
components
    // and returns the negated vector.
    public GM01.Vector3D negate(){
        return new GM01.Vector3D(new ColMatrix3D(
            -
getData(0),
            -
getData(1),
            -
getData(2)));
    } //end negate
    //-----
    -----//

    //Returns a new vector that points in the same
    // direction but has a length of one unit.
    public GM01.Vector3D normalize(){
        double length = getLength();

        return new GM01.Vector3D(new ColMatrix3D(
            getData(0)/length,
            getData(1)/length,
            getData(2)/length));
    }

```

```

    ~    }//end normalizé
        //-----
-----//
    }//end class Vector3D

//=====
===//

//=====
===//

```

```

    //A line is defined by two points. One is called
the
    // tail and the other is called the head. Note
that this
    // class has the same name as one of the classes
in
    // the Graphics2D class. Therefore, if the class
from
    // the Graphics2D class is used in some future
upgrade
    // to this program, it will have to be fully
qualified.
    public static class Line2D{
        GM01.Point2D[] line = new GM01.Point2D[2];

        public Line2D(GM01.Point2D tail,GM01.Point2D
head){
            //Create and save clones of the points used
to
            // define the line to prevent the line from
being
            // corrupted by a later change in the
coordinate
            // values of the points.
            this.line[0] = new Point2D(new

```



```

GM01.ColMatrix2D(
tail.getData(0),tail.getData(1)));
    this.line[1] = new Point2D(new
GM01.ColMatrix2D(
head.getData(0),head.getData(1)));
    }//end constructor
    //-----
    -----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + ","
            + line[1].getData(1);
    }//end toString
    //-----
    -----//

    public GM01.Point2D getTail(){
        return line[0];
    }//end getTail
    //-----
    -----//

    public GM01.Point2D getHead(){
        return line[1];
    }//end getHead
    //-----
    -----//

    public void setTail(GM01.Point2D newPoint){
        //Create and save a clone of the new point
to
        // prevent the line from being corrupted by
a

```

```

        // later change in the coordinate values of
the
        // point.
        this.line[0] = new Point2D(new
GM01.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
    }//end setTail
    //-----
-----//

    public void setHead(GM01.Point2D newPoint){
        //Create and save a clone of the new point
to
        // prevent the line from being corrupted by
a
        // later change in the coordinate values of
the
        // point.
        this.line[1] = new Point2D(new
GM01.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
    }//end setHead
    //-----
-----//

    public void draw(Graphics2D g2D){
        drawLine(g2D,getTail().getData(0),
                getTail().getData(1),
                getHead().getData(0),
                getHead().getData(1));
    }//end draw
    //-----
-----//
} //end class Line2D

```

```
//=====
====//
```

```
//A line is defined by two points. One is called  
the
```

```
// tail and the other is called the head.
```

```
public static class Line3D{
```

```
    GM01.Point3D[] line = new GM01.Point3D[2];
```

```
    public Line3D(GM01.Point3D tail,GM01.Point3D  
head){
```

```
        //Create and save clones of the points used  
to
```

```
        // define the line to prevent the line from  
being
```

```
        // corrupted by a later change in the  
coordinate
```

```
        // values of the points.
```

```
        this.line[0] = new Point3D(new  
GM01.ColMatrix3D(
```

```
tail.getData(0),
```

```
tail.getData(1),
```

```
tail.getData(2)));
```

```
        this.line[1] = new Point3D(new  
GM01.ColMatrix3D(
```

```
head.getData(0),
```

```
head.getData(1),
```

```
head.getData(2)));
```

```
    }//end constructor
```

```
    //-----
```

-----//

```
public String toString(){
    return "Tail = " + line[0].getData(0) + ","
        + line[0].getData(1) + ","
        + line[0].getData(2)
        + "\nHead = "
        + line[1].getData(0) + ","
        + line[1].getData(1) + ","
        + line[1].getData(2);
} //end toString
//-----
```

-----//

```
public GM01.Point3D getTail(){
    return line[0];
} //end getTail
//-----
```

-----//

```
public GM01.Point3D getHead(){
    return line[1];
} //end getHead
//-----
```

-----//

```
public void setTail(GM01.Point3D newPoint){
    //Create and save a clone of the new point
to    // prevent the line from being corrupted by
a    // later change in the coordinate values of
the    // point.
    this.line[0] = new Point3D(new
GM01.ColMatrix3D(
```

```

newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
    }//end setTail
    //-----
    -----//

    public void setHead(GM01.Point3D newPoint){
        //Create and save a clone of the new point
to        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the        // point.
        this.line[1] = new Point3D(new
GM01.ColMatrix3D(
newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
    }//end setHead
    //-----
    -----//

    public void draw(Graphics2D g2D){

        //Get 2D projection coordinates.
        GM01.ColMatrix2D tail =
convert3Dto2D(getTail().point);
        GM01.ColMatrix2D head =

```

```


convert3Dto2D(getHead().point);

        drawLine(g2D,tail.getData(0),
                  tail.getData(1),
                  head.getData(0),
                  head.getData(1));
    }//end draw
    //-----
-----//
} //end class Line3D

//=====
====//

} //end class GM01
//=====
=====//

```



Listing 31 . Source code for the game program named GM01test08.

```

/*GM01test08.java
Copyright 2008, R.G.Baldwin
Revised 02/28/08.

```

This is an interactive 3D game program This game simulates a school of prey fish being attacked and eaten by a predator. The objective for the player is to cause the predator to catch and eat all of the prey fish in a minimum amount of time.

The strategy is for the player to initiate a series of attacks by the predator, properly timed so as to minimize

the elapsed time required for the predator to catch and eat all of the prey fish. The final elapsed time depends both on player strategy and random chance. In other words, as is the case in many games, this game is part strategy and part chance.

The prey fish, (shown in red), swim in a large swirling cluster. The predator, (shown in blue), swims around the cluster, darting into the cluster on command to catch and eat as many prey fish as possible during each attack.

When the predator attacks, the prey fish tend to scatter, breaking up their tight formation and making it more difficult for the predator to catch them. However, if the predator is allowed to swim around them again for a short period following an attack, they will form back into a cluster. The attacks should be timed so that the prey fish are in a reasonably tight cluster at the beginning of each attack so that multiple prey fish will be caught and eaten during the attack. However, allowing too much time

between attacks is detrimental to minimizing the total elapsed time. Thus, the player must make a tradeoff between elapsed time between attacks and the tightness of the cluster during the attacks. Another disadvantage of waiting too long between attacks is explained below.

In addition to some other controls, the GUI provides an Attack button and an elapsed-time/kill-count indicator. At any point in time, this indicator displays the elapsed time in milliseconds when the most recent prey fish was caught along with the total number of prey fish that have been caught and eaten. The two values are separated by a "/" character.

When all of the prey fish have been caught and eaten by the predator, the elapsed time indicator shows the time in milliseconds required to catch and eat all of the prey fish. It also shows the number of prey fish that were caught and eaten, which should match one of the player input values.

Initially, when the player clicks the Start button, the predator is not in attack mode. The predator swims around the school of prey fish encouraging them to bunch up together in a smaller and tighter cluster. This behavior continues until the player clicks the Attack button, at which time the predator enters the attack mode and makes an attack on the cluster.

If the player clicks the Attack button too early, or doesn't wait long enough between attacks, the prey fish will be in a loose cluster, and the attack will yield very few fish.

If the player waits too long to click the Attack button, or waits too long between attacks, the predator will have spiraled in so close to the prey fish that they will break formation and begin to scatter, making it difficult for the predator to catch a large number of fish during the attack. This is the other disadvantage of waiting too long that I mentioned above.

The prey fish have a fairly effective defense

the prey fish have a fairly effective defense mechanism and can do a reasonably good job of escaping the predator.

The final three or four prey fish are usually the most difficult to catch.

When the predator is successful in catching a prey fish, that fish is removed from the prey-fish population, causing the prey-fish population to shrink over time.

Each time the predator is successful in catching a prey fish, the program emits an audible beep to provide feedback to the player.

Even when the predator is not in the attack mode, its presence has a significant effect on the behavior of the school of prey fish. As the predator swims around the school of prey fish, they tend to bunch up into a smaller and tighter cluster, but when the predator swims too close, the prey fish panic and tend to scatter, breaking up the tight cluster.

In addition to the elapsed time indicator and the Attack button, the GUI contains an input text field for the

...
number of prey fish that will be in the population
plus a
Start button and a Stop button. The GUI also
contains
check boxes that allow the player to display
points only,
direction vectors only, or both for the prey fish.
(Only
the direction vector is displayed for the
predator.)

The player specifies the number of randomly-placed
prey
fish that will be in the population and clicks the
Start
button to start the animation. At this point, the
prey
fish swim in a large swirling cluster and the
predator
swims around them encouraging them to form a
tighter
cluster. Prey fish motion is random but each fish
generally tends to spiral toward the center of the
cluster.

When the user clicks the Attack button, the
behavior is
as described earlier.

The animation continues until the user clicks the
Stop
button. The user can click the Stop button, change
any of
the parameters, and then click Start again to re-
start the
game with zero elapsed time and different
parameters.

The animation is most impressive when the direction

vectors are displayed for the prey fish because the vectors provide a lot of information about how the prey fish are reacting to the predator.

In addition to the school of prey fish and the predator, the graphical output also shows a large circle drawn with broken lines. This circle represents the intersection of a sphere and the x-y plane.

The purpose of the sphere, which is centered on the origin, is to provide a soft boundary for keeping the prey fish and the predator in the 3D playing field. The prey fish and the predator all have a tendency to remain inside the sphere, but they may occasionally stray outside the sphere. If they do, the program code will encourage them to return to the 3D playing field inside the sphere.

This game is fully 3D. The prey fish and the predator are free to swim in any direction in 3D space. The tails on the prey fish and the predator appear longest when

the prey, then and the predator appear longer when they are swimming parallel to the x-y plane. As they change

their angle relative to the x-y plane, the tails appear to become shorter and shorter until in some cases, they appear not to have a tail at all. This is best observed by clicking the Stop button just as the fish scatter during an attack and freezing the state of the fish in the 3D world. If you examine the image at that point, you are likely to see some fish with shorter tails than other fish.

Tested using JDK 1.6 under WinXP.

*****/

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.util.*;
```

```
class GM01test08{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class GM01test08
//=====
=====//
```

class GUI extends JFrame implements

```
-----
ActionListener{
```

```
    int hSize = 400;//horizontal size of JFrame.
    int vSize = 450;//vertical size of JFrame
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.
    int numberPoints = 0;//can be modified by the
user.
    JTextField numberPointsField; //user input
field.
    Random random = new Random();//random number
generator

    //The following collection is used to store the
prey
    // objects.
    ArrayList <GM01.Point3D> preyObjects;

    //The following collection is used to store the
vectors
    // for display.
    ArrayList <GM01.Vector3D> displayVectors;

    //The following are used to support user input.
    Checkbox drawPointsBox;//User input field
    Checkbox drawVectorsBox;//User input field.
    JButton attackButton;
    boolean drawPoints = true;
    boolean drawVectors = true;

    //The following JTextField is used to display
the
    // elapsed time and the number of prey fish that
    // have been caught and eaten.
```

```

//Have seen eagle and ocean
JTextField timer;

long baseTime;//Used to compute the elapsed
time.

//Used to compute the number of fish caught and
eaten.
int killCount = 0;

//Animation takes place while the following is
true.
boolean animate = false;

//Attacks take place while the following is
true.
boolean attack = false;

//Working variables used by the animation code.
GM01.Vector3D predatorVec;
GM01.Point3D preyCenter;
GM01.Point3D predator;
//-----
-----//

GUI(){//constructor

//Set JFrame size, title, and close operation.
setSize(hSize,vSize);
setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Instantiate the user input components.
numberPointsField = new JTextField("100",5);
JButton startButton = new JButton("Start");
attackButton = new JButton("Attack");
JButton stopButton = new JButton("Stop");

```

```

        timer = new JTextField("0",5);
        drawPointsBox = new Checkbox("Draw
Points",false);

        drawVectorsBox = new Checkbox("Draw
Vectors",true);

        //Instantiate a JPanel that will house the
user input
        // components and set its layout manager.
        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new GridLayout(0,2));

        //Add the user input component and appropriate
labels
        // to the control panel.
        controlPanel.add(new JLabel(" Number Prey
Fish"));
        controlPanel.add(numberPointsField);
        controlPanel.add(drawPointsBox);
        controlPanel.add(drawVectorsBox);
        controlPanel.add(startButton);
        controlPanel.add(attackButton);
        controlPanel.add(stopButton);
        controlPanel.add(timer);

        //Add the control panel to the SOUTH position
in the
        // JFrame.
        this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

        //Instantiate a new drawing canvas and add it
to the
        // CENTER of the JFrame above the control
panel.
        mvCanvas = new MvCanvas():

```



```

        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER, myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth, osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM01.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

        //Register this object as an action listener
on all
        // three buttons.
startButton.addActionListener(this);
attackButton.addActionListener(this);
stopButton.addActionListener(this);

        //Make the drawing color RED at startup.
g2D.setColor(Color.RED);

```

```
    }//end constructor
    //-----
-----//
```

```
    //This method is called to respond to a click on
any
```

```
    // of the three buttons.
    public void actionPerformed(ActionEvent e){

        if(e.getActionCommand().equals("Start") &&
!animate){
            //Service the Start button.
```

```
            //Get several user input values.
            numberPoints = Integer.parseInt(
numberPointsField.getText());
```

```
            if(drawPointsBox.getState()){
                drawPoints = true;
            }else{
                drawPoints = false;
            }//end else
```

```
            if(drawVectorsBox.getState()){
                drawVectors = true;
            }else{
                drawVectors = false;
            }//end else
```

```
            //Initialize some working variables used in
the
```

```
            // animation process.
            animate = true;
            baseTime = new Date().getTime();
            killCount = 0;
```

```

//Enable the Attack button.
attackButton.setEnabled(true);

//Initialize the text in the timer field.
timer.setText("0 / 0");

//Cause the run method belonging to the
animation
// thread object to be executed.
new Animate().start();
} //end if

if(e.getActionCommand().equals("Attack") &&
animate){
//Service the Attack button.

//This will cause the predator to attack the
prey
// objects.
attack = true;

//Point the predator toward the center of
the prey
// fish formation. Reduce the scale factor
to less
// than 1.0 to slow down the attack and make
it
// easier for the prey fish to escape the
predator.
predatorVec =
predator.getDisplacementVector(preycenter).
scale(1.0);

//Disable the Attack button. It will be

```

```

        enabled
        // again when the attack is finished.
        attackButton.setEnabled(false);
    }//end if

    if(e.getActionCommand().equals("Stop") &&
animate){
        //Service the Stop button.

        //This will cause the run method to
terminate and
        // stop the animation. It will also clear
the
        // attack flag so that the next time the
animation
        // is started, the predator won't be in the
        // attack mode.
        animate = false;
        attack = false;
    }//end if

} //end actionPerformed

//=====
====//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the update method to eliminate the
default
    // clearing of the Canvas in order to reduce
or
    // eliminate the flashing that that is often
caused by
    // such default clearing.
    //In this case. it isn't necessary to clear

```

```

        // In this case, it is not necessary to show
the canvas
        // because the off-screen image is cleared
each time

        // it is updated. This method will be called
when the
        // JFrame and the Canvas appear on the screen
or when
        // the repaint method is called on the Canvas
object.
        public void update(Graphics g){
            paint(g); //Call the overridden paint method.
        } //end overridden update()

        //Override the paint() method. The purpose of
the
        // paint method is to display the off-screen
image on
        // the screen. This method is called by the
update
        // method above.
        public void paint(Graphics g){
            g.drawImage(osi, 0, 0, this);
        } //end overridden paint()

    } //end inner class MyCanvas

//=====
====//

    //This is an animation thread. The purpose of
this
    // animation is to simulate something like a
school of
    // prey fish and a predator. The animation
behavior
    // is described in the opening comments for the
program

```

```

// as a whole.
class Animate extends Thread{
    //Declare a general purpose variable of type
Point3D.
    // It will be used for a variety of purposes.
    GM01.Point3D tempPrey;

    //Declare two general purpose variables of
type
    // Vector3D. They will be used for a variety
of
    // purposes.
    GM01.Vector3D tempVectorA;
    GM01.Vector3D tempVectorB;
    //-----
-----//

    public void run(){
        //This method is executed when start is
called on
        // this Thread object.
        //Create a new empty container for the prey
objects.
        //Note the use of "Generics" syntax.
        preyObjects = new ArrayList<GM01.Point3D>();

        //Create a new empty container for the
vectors. The
        // only reason the vectors are saved is so
that they
        // can be displayed later
        displayVectors = new
ArrayList<GM01.Vector3D>();

        //Create a set of prey objects at random
locations
        // and store references to the prey objects

```

```

in the
    // preyObjects container.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        preyObjects.add(new GM01.Point3D(
            new GM01.ColMatrix3D(
                100*
(random.nextDouble()-0.5),
                100*
(random.nextDouble()-0.5),
                100*
(random.nextDouble()-0.5))));

        //Populate the displayVectors collection
with
        // dummy vectors.
        displayVectors.add(tempVectorA);
    }//end for loop

    //This object that will be the predator.
    // Position it initially near the top of the
screen.
    predator = new GM01.Point3D(
        new
GM01.ColMatrix3D(-100,100,-100));

    //Create a reference point to mark the
origin. Among
    // other things, it will be used as the
center of a
    // sphere, which in turn will be used to
attempt to
    // keep the prey and predator objects from
leaving
    // the playing field.
    GM01.Point3D origin =
        new GM01.Point3D(new
GM01.ColMatrix3D(0.0,0.0,0.0));

```

.....\,/,/,/,/,/

```
        //Declare some variables that will be used
to
        // compute and save the average position of
all of
        // the prey objects.
        double xSum = 0;
        double ySum = 0;
        double zSum = 0;

        //This is the animation loop. The value of
animate
        // is set to true by the Start button and is
set to
        // false by the Stop button. Setting it to
false
        // will cause the run method to terminate
and stop
        // the animation.
        while(animate){
            //Compute and save the average position of
all the
            // prey objects at the beginning of the
loop. Save
            // the average position in the Point3D
object
            // referred to by preyCenter.
            xSum = 0;
            ySum = 0;
            zSum = 0;

            for(int cnt = 0;cnt <
preyObjects.size();cnt++){
                tempPrey = preyObjects.get(cnt);
                xSum += tempPrey.getData(0);
                vSum += tempPrey.getData(1);
```



```

        zSum += tempPrey.getData(2);
    } //end for loop

    //Construct a reference point at the
average
    // position of all the prey objects.
    preyCenter = new GM01.Point3D(
        new
GM01.ColMatrix3D(xSum/preyObjects.size(),
ySum/preyObjects.size(),
zSum/preyObjects.size()));

    //Move all of the prey objects in a way
that
    // causes them to spiral toward the
preyCenter.
    for(int cnt = 0; cnt <
preyObjects.size(); cnt++){
        //Stop spiraling toward the center when
the
        // population of prey objects contains a
        // single object because there is no
center at
        // that point.
        if(preObjects.size() > 1){
            //Get the next prey object
            tempPrey = preyObjects.get(cnt);
            //Find the displacement vector from
this prey
            // object to the preyCenter
            tempVectorA =
tempPrey.getDisplacementVector(
preyCenter):

```

```

//Create a vector that is rotated
relative to
// tempVectorA. Note how each
component in
// this new vector is set equal to a
different
// component in tempVectorA. Scale it.
tempVectorB = new GM01.Vector3D(
    new GM01.ColMatrix3D(
        tempVectorA.getData(1),
        tempVectorA.getData(2),
tempVectorA.getData(0))).scale(1.2);

//Add the two vectors and move the
prey object
// according to the sum of the
vectors. Scale
// one of the vectors during the
addition to
// adjust the rate at which the object
spirals
// toward the center.
//Moving the prey object in the
direction of
// the sum of these two vectors
produces a
// motion that causes the prey object
to
// spiral toward the preyCenter
instead of
// simply moving in a straight line
toward the
// preyCenter.
tempVectorA =

```

```

tempVectorA.scale(0.9).add(tempVectorB);
    //Apply an overall scale factor to the
sum
    // vector before using it to move the
prey
    // object.
    tempPrey = tempPrey.addVectorToPoint(
tempVectorA.scale(0.1));

    //Save a clone of the prey object in
its new
    // position. Save a clone instead of
the
    // actual object as a safety measure
to avoid
    // the possibility of corrupting the
object
    // later when the reference variable
is used
    // for some other purpose.
    preyObjects.set(cnt,tempPrey.clone());

    //Save a normalized version of the
direction
    // vector for drawing later. Set the
length
    // of the normalized vector to 15
units.
    displayVectors.set(
cnt,tempVectorA.normalize().scale(15.0));
    }else{
    //When the population consists of a
single
    // prey object, save a dummy direction
vector

```

```

.....
        // for drawing later. This is
necessary to
        // prevent a null pointer exception
when the
        // user specifies only one prey object
and
        // then clicks the Start button.
displayVectors.set(
            cnt,new GM01.Vector3D(
                new
GM01.ColMatrix3D(10,10,0)));
        }//end else

    }//end loop to spiral prey objects toward
center.

        //Try to keep the prey objects from
colliding with
        // one another by moving each prey object
away
        // from its close neighbors.
GM01.Point3D refPrey = null;
GM01.Point3D testPrey= null;
for(int row = 0;row <
preyObjects.size();row++){
    refPrey = preyObjects.get(row);
    //Compare the position of the reference
prey
    // object with the positions of each of
the
    // other prey objects.
    for(int col = 0;col <
preyObjects.size();col++){
        //Get another prey object for
proximity test.
        testPrev = preyObjects.get(col):

```

```

        //Don't test a prey object against
itself.
        if(col != row){
            //Get the vector from the refPrey
object to
            // the testPrey object.
            tempVectorA = refPrey.
getDisplacementVector(testPrey);

            //If refPrey is too close to
testPrey, move
            // it away from the testPrey object.
            if(tempVectorA.getLength() < 10){
                //Move refPrey away from testPrey
by a
                // small amount in the opposite
direction.
                refPrey =
refPrey.addVectorToPoint(
tempVectorA.scale(0.2).negate());

            }//end if on proximity test
        }//end if col != row
    }//end loop on col
    //The refPrey object may or may not have
been
    // moved. Save it anyway.
    preyObjects.set(row,refPrey.clone());
} //end loop on row

//Make each prey object react to the
predator with
// a defensive mechanism when the predator
// approaches the prey object.

```

```

// approach the prey object
for(int cnt = 0;cnt <
preyObjects.size();cnt++){
    tempPrey = preyObjects.get(cnt);

    //Get a displacement vector from the
prey object
    // to the predator.
    tempVectorA =
tempPrey.getDisplacementVector(
predator);

    //If the prey object is within a certain
    // threshold distance from the predator,
move
    // the prey object a random distance in
the
    // opposite direction. Then test again
to see if
    // the prey object is still within
another
    // smaller threshold distance. If so,
the prey
    // object was not successful in escaping
the
    // predator and has been caught and
eaten by the
    // predator. Remove the prey object from
the
    // population and sound a beep to notify
the
    // user of the successful attack by the
    // predator.
    if(tempVectorA.getLength() < 50){
        //The predator is within striking
range of the
        // prey object. Make the prey object

```

```

try to
    // escape by moving a random distance
in the
    // opposite direction. The value
returned by
    // the random number generator ranges
from 0
    // to 1.0.
    tempVectorA =
tempVectorA.negate().scale(
random.nextDouble());
    tempPrey =
tempPrey.addVectorToPoint(tempVectorA);

    //Check to see if the escape was
successful
    // by testing against a smaller
threshold
    // distance.
    //Get a new displacement vector from
the prey
    // object to the predator.
    tempVectorA =
tempPrey.getDisplacementVector(predator);
    //Decrease or increase the following
    // threshold distance to cause the
prey
    // object to be more or less
successful in the
    // escape attempt.
    if(tempVectorA.getLength() < 25){
        //Escape was not successful. The
predator is
        // still within striking distance of

```

the
object and
population
the
display

milliseconds
clicked along
been
is
when the
by the
elapsed
clicked.
caught, the
the time
catch all of

```
// ===== Remove the prey object. Remove the prey
// its direction vector from the
// and sound a beep. Also increase
// killCount by 1 for purposes of
// only.
tempPrey = preyObjects.remove(cnt);
displayVectors.remove(cnt);
Toolkit.getDefaultToolkit().beep();
killCount++;

//Display the elapsed time in
// since the Start button was
// with the number of fish that have
// caught and eaten. The value that
// displayed is the elapsed time
// most recent prey fish was caught
// predator and is not the total
// time since the Start button was
// When all prey fish have been
// final time that is displayed is
// required for the predator to
```



```

// the fish in the population.
timer.setText(
    ""
    + (new Date().getTime() -
baseTime)
    + " / "
    + killCount);

}else{
    //The escape attempt was successful.
Restore
    // a clone of the prey object in its
new
    // location along with its direction
vector
    // to the population.

preyObjects.set(cnt,tempPrey.clone());
    //Save a normalized version of the
direction
    // vector for drawing later. It
will
    // overwrite the previously saved
vector and
    // if you watch closely it will show
the
    // prey object running away from the
    // predator..
    displayVectors.set(
cnt,tempVectorA.normalize().scale(15.0));
    }//end else
    }//end if distance < 50
}//end for loop on population

//Deal with prev objects that strav

```

```

// move them prey objects that are
outside the
// spherical boundary. Give them a nudge
back
// toward the origin.
for(int cnt = 0;cnt <
preyObjects.size();cnt++){
    tempPrey = preyObjects.get(cnt);

    //Test to determine if the prey object
is
    // outside of a sphere centered on the
origin
    // with a radius equal to the maximum
dimension
    // on the vertical axis.

if(tempPrey.getDisplacementVector(origin).
                                getLength() >
0.5*osiHeight){
    //Enable the following statement to
get an
    // audible beep each time a prey
object goes
    // out of bounds.
    //Toolkit.getDefaultToolkit().beep();
    //Give the prey object a nudge back
toward the
    // origin and save a clone of the prey
object
    // in its new location.
    tempVectorA =

tempPrey.getDisplacementVector(origin);
    tempPrey = tempPrey.addVectorToPoint(
tempVectorA.scale(0.1));
    preyObjects.set(cnt,tempPrey.clone());

```

```

        }//end if prey object is out of bounds
    }//end for loop to process each prey
object

```

```

//Erase the screen
g2D.setColor(Color.WHITE);
GM01.fillRect(g2D, -osiWidth/2, osiHeight/2,
osiWidth, osiHeight);

```

```

//Draw a broken-line circle that
represents the
// intersection of the spherical boundary
with the
// x-y plane. Although the prey objects
and the
// predator can stray outside this sphere,
they
// prefer to be inside the sphere.
GM01.Point3D tempPointA = new
GM01.Point3D(
    new
GM01.ColMatrix3D(osiHeight/2, 0, 0));
GM01.Point3D tempPointB;
//The circle is defined by 39 points
around the
// circumference.
g2D.setColor(Color.BLACK);
for(int cnt = 0; cnt < 39; cnt++){
    tempPointB =
        new GM01.Point3D(new GM01.ColMatrix3D(
osiHeight/2*Math.cos((cnt*360/39)*Math.PI/180),
osiHeight/2*Math.sin((cnt*360/39)*Math.PI/180).

```

```

        0));

        //Connect every third pair of points
with a line
        if(cnt%3 == 0){
            new GM01.Line3D(
tempPointA,tempPointB).draw(g2D);
        }//end if
        //Save the old point.
        tempPointA = tempPointB;
    }//end for loop

    //Draw the final line required to close
the circle
        new GM01.Line3D(tempPointA,new
GM01.Point3D(
                                new GM01.ColMatrix3D(
osiHeight/2,0,0))).draw(g2D);

        //Restore the drawing color.
        g2D.setColor(Color.RED);
        //Draw the objects in the preyObjects
container
        // and the vectors in the displayVectors
        // container.
        for(int cnt = 0;cnt <
preyObjects.size();cnt++){
            tempPrey = preyObjects.get(cnt);
            if(drawPoints){
                tempPrey.draw(g2D);//draw circle
around point
            }//end if

            if(drawVectors){

```

```
        //Draw the vector with its tail at the  
point.
```

```
displayVectors.get(cnt).draw(g2D,tempPrey);  
    }//end if  
}//end for loop
```

```
    //When the predator is not in attack mode,  
cause
```

```
    // it to slowly circle the cluster of prey  
    // objects.  
    if(!attack){  
        //Get a displacement vector pointing  
from the  
        // predator to the preyCenter.  
        predatorVec =
```

```
predator.getDisplacementVector(preycCenter);
```

```
        //Create a vector that is rotated  
relative to  
        // predatorVec. Note how each component  
in  
        // this new vector is set equal to a  
different
```

```
        // component in predatorVec  
        tempVectorB = new GM01.Vector3D(  
            new GM01.ColMatrix3D(  
                predatorVec.getData(1),  
                predatorVec.getData(2),  
predatorVec.getData(0))).scale(0.15);
```

```
        //Scale predatorVec and add the two  
vectors.  
        // Then move the predator according to
```

```

the sum    // then move the predator according to
           // of the vectors.
           //Moving the prey object in the
direction of
           // the sum of these two vectors produces
a
           // motion that causes the predator to
           // spiral toward the preyCenter instead
of
           // simply moving in a straight line
toward the
           // preyCenter. The scale factors control
the
           // relative motion between the two
directions,
           // and are fairly sensitive.
           predatorVec =

predatorVec.scale(0.095).add(tempVectorB);

           predator = predator.addVectorToPoint(

predatorVec.scale(1.0));

           }else{//attack is true
           //Predator is in attack mode now. Stay
in attack
           // mode using the same displacement
vector until
           // the predator traverses the entire
playing
           // field. This displacement vector
originally
           // pointed toward the preyCenter and was
created
           // in the actionPerformed method in
response to

```

```

// a click on the Attack button. In
other words,
// even though the prey fish scatter,
the
// predator is constrained to move in a
straight
// line across the playing field once an
attack
// is begun. An interesting update might
be to
// allow the predator to adjust its
direction
// as the prey fish scatter and the
location of
// preyCenter changes during the
traversal
// across the playing field.
predator = predator.addVectorToPoint(
predatorVec.scale(0.25));

//Check to see if the predator is
outside the
// spherical boundary that defines the
playing
// field.

if(predator.getDisplacementVector(origin).
getLength() >
0.5*osiHeight){
//Shift out of attack mode and start
circling
// the prey fish again.
attack = false;
attackButton.setEnabled(true);
} //end if
} //end else

```

```

//Set the color to BLUE and draw the
predator and
    // its displacement vector.
    g2D.setColor(Color.BLUE);
    //Enable the following statement to draw a
circle
    // around the point that represents the
predator.
    //predator.draw(g2D);

    //Draw the predator's vector.
    predatorVec.normalize().scale(15.0).draw(
g2D,predator);
    g2D.setColor(Color.RED); //restore red
color

    //Copy the off-screen image to the canvas
and then
    // do it all again.
    myCanvas.repaint();

    //Insert a time delay. Change the sleep
time to
    // speed up or slow down the animation.
    try{
        Thread.currentThread().sleep(166);
    }catch(Exception e){
        e.printStackTrace();
    } //end catch
} //end animation loop
} //end run method
} //end inner class named Animate

//=====

```



```
...  
====//  
  
} //end class GUI
```

Listing 32 . Source code for the program named GM01test04.

```
/*GM01test04.java  
Copyright 2008, R.G.Baldwin  
Revised 02/25/08.
```

This animation program is designed to test many of the 2D features of the GM01 game-math library.

The animation is generally based on the idea of a flocking behavior similar to that exhibited by birds.

A set of Point2D objects is created with random locations.
An additional Point2D object known as the target is also created. The target is drawn in red while the pursuers are drawn in black.

An algorithm is executed that attempts to cause the points to chase the target without colliding with one another.
Even though the algorithm causes the points to chase the target, it also tries to keep the points from colliding with the target.

A GUI is provided that contains an input text field for the number of points plus a Start button and a Stop button. The GUI also contains check boxes that allow the user to elect to display points only, direction vectors only, or both. (The point and the direction vector is always displayed for the target.)

The user specifies the number of randomly-placed points and clicks the Start button, at which time the animation begins and the points start chasing the target. Target motion is random. The animation continues until the user clicks the Stop button. The user can click the Stop button, change the number of points, and then click Start again to re-start the animation with a different number of points chasing the target.

The algorithm seems to cause the points to come together and fly in formation while chasing the target. The most common formation is hexagonal but sometimes triangles and incomplete hexagons also appear. I have no explanation for this behavior. The tendency to fly in formation is

more
visually obvious when only the points are
displayed.

On the other hand, the animation is most
impressive when
the direction vectors are displayed, with or
without
points, because the vectors illustrate the dogged
determination and undying focus that the pursuers
maintain
while chasing the target.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/
```

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.geom.*;  
import java.awt.event.*;  
import java.util.*;
```

```
class GM01test04{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class GM01test04  
//=====
```

```
class GUI extends JFrame implements  
ActionListener{
```

```
    int hSize = 400;//horizontal size of JFrame.  
    int vSize = 400;//vertical size of JFrame  
    Image osi;//an off-screen image  
    int osiWidth;//off-screen image width
```

```

    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.
    int numberPoints = 0;//can be modified by the
user.
    JTextField numberPointsField; //user input
field.
    Random random = new Random();//random number
generator

    //The following collection is used to store the
points.
    ArrayList <GM01.Point2D> points;

    //The following collection is used to store the
vectors
    // for display.
    ArrayList <GM01.Vector2D> vectors;

    //The following are used to support user input.
    Checkbox drawPointsBox;//User input field
    Checkbox drawVectorsBox;//User input field.
    boolean drawPoints = true;
    boolean drawVectors = true;

    //Animation takes place while the following is
true.
    boolean animate = false;
    //-----
    -----//

    GUI(){//constructor

        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

```

```

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate the user input components.
    numberPointsField =new JTextField("6",5);
    JButton startButton = new JButton("Start");
    JButton stopButton = new JButton("Stop");
    drawPointsBox = new Checkbox("Draw
Points",true);
    drawVectorsBox = new Checkbox("Draw
Vectors",true);

    //Instantiate a JPanel that will house the
user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,2));

    //Add the user input component and appropriate
labels
    // to the control panel.
    controlPanel.add(new JLabel(" Number
Points"));
    controlPanel.add(numberPointsField);
    controlPanel.add(drawPointsBox);
    controlPanel.add(drawVectorsBox);
    controlPanel.add(startButton);
    controlPanel.add(stopButton);

    //Add the control panel to the SOUTH position
in the
    // JFrame.
    this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

    //Instantiate a new drawing canvas and add it
to the

```

```

        // CENTER of the JFrame above the control
panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER, myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth, osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM01.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

        //Register this object as an action listener
on the
        // startButton and the stopButton
startButton.addActionListener(this);
stopButton.addActionListener(this);

} //end constructor

```

```
//-----  
-----//
```

```
//This method is called to respond to a click on  
the
```

```
// startButton or the stopButton.
```

```
public void actionPerformed(ActionEvent e){
```

```
    if(e.getActionCommand().equals("Start") &&  
!animate){
```

```
        numberPoints = Integer.parseInt(
```

```
numberPointsField.getText());
```

```
    if(drawPointsBox.getState()){
```

```
        drawPoints = true;
```

```
    }else{
```

```
        drawPoints = false;
```

```
    }//end else
```

```
    if(drawVectorsBox.getState()){
```

```
        drawVectors = true;
```

```
    }else{
```

```
        drawVectors = false;
```

```
    }//end else
```

```
    animate = true;
```

```
    new Animate().start();
```

```
}//end if
```

```
    if(e.getActionCommand().equals("Stop") &&  
animate){
```

```
        //This will cause the run method to  
terminate and
```

```
        // stop the animation.
```

```
        animate = false;
```

```
    }//end if
```

```

    }//end actionPerformed

//=====
====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the update method to eliminate the
default
        // clearing of the Canvas in order to reduce
or
        // eliminate the flashing that that is often
caused by
        // such default clearing.
        //In this case, it isn't necessary to clear
the canvas
        // because the off-screen image is cleared
each time
        // it is updated. This method will be called
when the
        // JFrame and the Canvas appear on the screen
or when
        // the repaint method is called on the Canvas
object.
        public void update(Graphics g){
            paint(g);//Call the overridden paint method.
        }//end overridden update()

        //Override the paint() method. The purpose of
the
        // paint method is to display the off-screen
image on
        // the screen. This method is called by the
update
        // method above.
        public void paint(Graphics g){

```



```

        g.drawImage(osi, 0, 0, this);
    }//end overridden paint()

}//end inner class MyCanvas

//=====
====//

    //This is an animation thread.
    class Animate extends Thread{
        //Declare a general purpose variable of type
Point2D.
        // It will be used for a variety of purposes.
        GM01.Point2D tempPoint;

        //Declare a general purpose variable of type
Vector2D.
        // It will be used for a variety of purposes.
        GM01.Vector2D tempVector;

        public void run(){
            //This method is executed when start is
called on
            // this thread.
            //Create a new empty container for the
points.
            points = new ArrayList<GM01.Point2D>();

            //Create a new empty container for the
vectors.
            vectors = new ArrayList<GM01.Vector2D>();

            //Create a set of Point objects at random
locations
            // and store references to the points in the
            // ArrayList object..
            for(int cnt = 0; cnt < numberPoints; cnt++){

```

```

        points.add(new GM01.Point2D(
            new GM01.ColMatrix2D(
                100*
(random.nextDouble()-0.5),
                100*
(random.nextDouble()-0.5))));

        //Populate vectors collection with dummy
vectors.
        vectors.add(tempVector);
    }//end for loop

    //Create a Point2D object that will be the
target
    // that will be chased by the other Point2D
objects.
    GM01.Point2D target = new GM01.Point2D(
        new GM01.ColMatrix2D(
            100*
(random.nextDouble()-0.5),
            100*
(random.nextDouble()-0.5))));

    //Create a Vector2D object that will be used
to
    // control the motion of the target.
    GM01.Vector2D targetVec = new GM01.Vector2D(
        new GM01.ColMatrix2D(
            100*(random.nextDouble()-0.5),
            100*
(random.nextDouble()-0.5))).scale(0.3);

    //Create a reference point to mark the
origin.
    GM01.Point2D zeroPoint =
        new GM01.Point2D(new
GM01.ColMatrix2D(0,0));

```

```

        //Declare a variable that will be used to
control
        // the update frequency of the target
vector.
        int animationLoopCounter = 0;

        //This is the animation loop. The value of
animate
        // is set to true by the Start button and is
set to
        // false by the Stop button. Setting it to
false
        // will cause the run method to terminate
and stop
        // the animation.
        while(animate){
            animationLoopCounter++;

            //Try to keep the points from colliding
with one
            // another. Note, however, that this
algorithm
            // is far from perfect in accomplishing
that.
            for(int row = 0;row < points.size();row++)
            {
                for(int col = 0;col <
points.size();col++){
                    GM01.Point2D refPoint =
points.get(row);
                    GM01.Point2D testPoint =
points.get(col);

                    if(col != row){
                        //Get the distance of the testPoint
from the

```

```

        // refPoint.
        tempVector = testPoint.

getDisplacementVector(refPoint);

        if(tempVector.getLength() < 25){
            //Modify testPoint to move it away
from
            // the refPoint and save the
modified
            // point.
            tempVector = tempVector.negate();
            tempPoint =
testPoint.addVectorToPoint(
tempVector.scale(0.1));
            points.set(col,tempPoint.clone());
        }else{
            //Do nothing.
        } //end else
    } //end if col != row
} //end loop on col
} //end loop on row

//Move all of the points toward the target
but try
// to keep them separated from the target.
for(int cnt = 0;cnt < points.size();cnt++)
{
    //Get the next point.
    tempPoint = points.get(cnt);
    //Find the distance from this point to
the
    // target.
    tempVector =

tempPoint.getDisplacementVector(target);

```

```

        if(tempVector.getLength() < 10){
            //Modify the point to move it away
from the
            // target and save the modified point.
            tempVector = tempVector.negate();
            tempPoint =
tempPoint.addVectorToPoint(
tempVector.scale(0.1));
            //Save the modified point.
            points.set(cnt,tempPoint.clone());
        }else{
            //Modify the point to move it toward
the
            // target and save the modified point.
            tempPoint =
tempPoint.addVectorToPoint(
tempVector.scale(0.1));
            points.set(cnt,tempPoint.clone());
        }//end else

        //Save a normalized version of the
direction
        // vector for drawing later.
        vectors.set(
cnt,tempVector.normalize().scale(15.0));

    }//end for loop

    //Insert a delay.
    try{
        Thread.currentThread().sleep(166);
    }catch(Exception e){
        e.printStackTrace();
    }//end catch

```

```

        //Erase the screen
        g2D.setColor(Color.WHITE);
        GM01.fillRect(g2D, -osiWidth/2,osiHeight/2,
osiWidth,osiHeight);
        g2D.setColor(Color.BLACK);

        //Draw the points and the vectors in the
        arrayList
        // objects.
        for(int cnt = 0;cnt < numberPoints;cnt++){
            tempPoint = points.get(cnt);
            if(drawPoints){
                tempPoint.draw(g2D);
            }//end if

            if(drawVectors){
                tempVector = vectors.get(cnt);
                tempVector.draw(g2D,tempPoint);
            }//end if
        }//end for loop


        //Cause the targets movements to be
        slightly less
        // random by using the same vector for
        several
        // successive movements
        if(animationLoopCounter%10 == 0){
            //Get a new vector
            targetVec = new GM01.Vector2D(
                new GM01.ColMatrix2D
                (100*(random.nextDouble()-0.5),
                100*
                (random.nextDouble()-0.5))).scale(0.3);
        }//else use the same vector again.

```

```

        //Test to see if this displacement vector
will    // push the target outside the limit. If
        // modify the vector to send the target
so,      back
        // toward the origin.

if(target.getDisplacementVector(zeroPoint).
                                getLength() >
0.4*osiWidth){
        targetVec =
target.getDisplacementVector(
zeroPoint).scale(0.1);

        }//end if

        //Modify the location of the target point.
        target =
target.addVectorToPoint(targetVec);

        //Set the color to RED and draw the target
and its // vector.
        g2D.setColor(Color.RED);
        target.draw(g2D);
        targetVec.normalize().scale(15.0).draw(
g2D,target);

        g2D.setColor(Color.BLACK);

        myCanvas.repaint();
    }//end while loop
} //end run

```

```

    }//end inner class named Animate

//=====
====//

}//end class GUI

```

Listing 33 . Source code for the program named GM01test03.

```

/*GM01test03.java
Copyright 2008, R.G.Baldwin
Revised 02/25/08.

```

This is a 3D update from the 2D version named GM01test04.
Basically all that was required to perform the update was
to specify 3D classes from the game-math library in place
of the 2D classes used in the 2D version.
Virtually everything else took care of itself.

The visual behavior of this 3D version is more realistic
than the 2D version. This is particularly true when the
target gets in the middle of the pursuers and the display
is showing both points and vectors. In this 3D version,
a pursuer is free to swing around in any plane as the
target passes by whereas in the 2D version, a pursuer is
constrained to swing around only in the plane of

the
screen. That causes the motion to be less fluid.
This is
best demonstrated with only one pursuer because
that makes
it easy to see the behavior of an individual
pursuer.

One thing that I did notice, is that unlike the 2D
version, the pursuers in this 3D version don't
seem to
have a tendency to form up and fly in formation
while
chasing the target. This may be because they have
more
options in terms of avoiding collisions while
giving
chase, but that is pure speculation since I don't
know
why they tend to fly in formation in the 2D
version.

This animation program is designed to test many of
the 3D
features of the GM01 game-math library.

The animation is generally based on the idea of a
flocking behavior similar to that exhibited by
birds.

A set of Point3D objects is created with random
locations.
An additional Point3D object known as the target
is also
created. The target is drawn in red while the
pursuers
are drawn in black.

An algorithm is executed that attempts to cause the points to chase the target without colliding with one another. Even though the algorithm causes the points to chase the target, it also tries to keep the points from colliding with the target.

A GUI is provided that contains an input text field for the number of points plus a Start button and a Stop button. The GUI also contains check boxes that allow the user to elect to display points only, direction vectors only, or both. (Both the point and the direction vector are always displayed for the target.)

The user specifies the number of randomly-placed points and clicks the Start button, at which time the animation begins and the points start chasing the target. Target motion is random. The animation continues until the user clicks the Stop button. The user can click the Stop button, change the number of points, and then click Start again to re-start the animation with a different number

of points chasing the target.

The animation is most impressive when the direction vectors are displayed, with or without points, because the vectors illustrate the dogged determination and undying focus that the pursuers maintain while chasing the target.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/
```

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.geom.*;  
import java.awt.event.*;  
import java.util.*;
```

```
class GM01test03{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class GM01test03  
//=====
```

```
class GUI extends JFrame implements  
ActionListener{
```

```
    int hSize = 400;//horizontal size of JFrame.  
    int vSize = 400;//vertical size of JFrame  
    Image osi;//an off-screen image  
    int osiWidth;//off-screen image width  
    int osiHeight;//off-screen image height  
    MyCanvas myCanvas;//a subclass of Canvas
```

```

    Graphics2D g2D; //off-screen graphics context.
    int numberPoints = 0; //can be modified by the
user.
    JTextField numberPointsField; //user input
field.
    Random random = new Random(); //random number
generator

    //The following collection is used to store the
points.
    ArrayList <GM01.Point3D> points;

    //The following collection is used to store the
vectors

    // for display.
    ArrayList <GM01.Vector3D> vectors;

    //The following are used to support user input.
    Checkbox drawPointsBox; //User input field
    Checkbox drawVectorsBox; //User input field.
    boolean drawPoints = true;
    boolean drawVectors = true;

    //Animation takes place while the following is
true.
    boolean animate = false;
    //-----
    -----//

    GUI(){//constructor

        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```

        //Instantiate the user input components.
        numberPointsField =new JTextField("6",5);
        JButton startButton = new JButton("Start");
        JButton stopButton = new JButton("Stop");
        drawPointsBox = new Checkbox("Draw
Points",true);
        drawVectorsBox = new Checkbox("Draw
Vectors",true);

        //Instantiate a JPanel that will house the
user input
        // components and set its layout manager.
        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new GridLayout(0,2));

        //Add the user input component and appropriate
labels
        // to the control panel.
        controlPanel.add(new JLabel(" Number
Points"));
        controlPanel.add(numberPointsField);
        controlPanel.add(drawPointsBox);
        controlPanel.add(drawVectorsBox);
        controlPanel.add(startButton);
        controlPanel.add(stopButton);

        //Add the control panel to the SOUTH position
in the
        // JFrame.
        this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

        //Instantiate a new drawing canvas and add it
to the
        // CENTER of the JFrame above the control
panel.

```

```

        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER, myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth, osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM01.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

        //Register this object as an action listener
on the
        // startButton and the stopButton
startButton.addActionListener(this);
stopButton.addActionListener(this);

    } //end constructor
    //-----
    -----//

```

```

    //This method is called to respond to a click on
the
    // startButton or the stopButton.
    public void actionPerformed(ActionEvent e){

        if(e.getActionCommand().equals("Start") &&
!animate){
            numberPoints = Integer.parseInt(
numberPointsField.getText());

            if(drawPointsBox.getState()){
                drawPoints = true;
            }else{
                drawPoints = false;
            }//end else

            if(drawVectorsBox.getState()){
                drawVectors = true;
            }else{
                drawVectors = false;
            }//end else

            animate = true;
            new Animate().start();
        }//end if

        if(e.getActionCommand().equals("Stop") &&
animate){
            //This will cause the run method to
terminate and
            // stop the animation.
            animate = false;
        }//end if

    }//end actionPerformed

```

```
//=====
====//
```

```
//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the update method to eliminate the
default
    // clearing of the Canvas in order to reduce
or
    // eliminate the flashing that that is often
caused by
    // such default clearing.
    //In this case, it isn't necessary to clear
the canvas
    // because the off-screen image is cleared
each time
    // it is updated. This method will be called
when the
    // JFrame and the Canvas appear on the screen
or when
    // the repaint method is called on the Canvas
object.
    public void update(Graphics g){
        paint(g); //Call the overridden paint method.
    } //end overridden update()

    //Override the paint() method. The purpose of
the
    // paint method is to display the off-screen
image on
    // the screen. This method is called by the
update
    // method above.
    public void paint(Graphics g){
        g.drawImage(osi, 0, 0, this);
    } //end overridden paint()
```



```

    }//end inner class MyCanvas

//=====
====//

    //This is an animation thread.
    class Animate extends Thread{
        //Declare a general purpose variable of type
Point3D.
        // It will be used for a variety of purposes.
        GM01.Point3D tempPoint;

        //Declare a general purpose variable of type
Vector3D.
        // It will be used for a variety of purposes.
        GM01.Vector3D tempVector;

        public void run(){
            //This method is executed when start is
called on
            // this thread.
            //Create a new empty container for the
points.
            points = new ArrayList<GM01.Point3D>();

            //Create a new empty container for the
vectors.
            vectors = new ArrayList<GM01.Vector3D>();

            //Create a set of Point objects at random
locations
            // and store references to the points in the
            // ArrayList object..
            for(int cnt = 0;cnt < numberPoints;cnt++){
                points.add(new GM01.Point3D(
                    new GM01.ColMatrix3D(

```

```

                                100*
(random.nextDouble()-0.5),
                                100*
(random.nextDouble()-0.5),
                                100*
(random.nextDouble()-0.5))));

        //Populate vectors collection with dummy
vectors.
        vectors.add(tempVector);
    }//end for loop

    //Create a Point3D object that will be the
target
    // that will be chased by the other Point3D
objects.
    GM01.Point3D target = new GM01.Point3D(
        new GM01.ColMatrix3D(
                                100*
(random.nextDouble()-0.5),
                                100*
(random.nextDouble()-0.5),
                                100*
(random.nextDouble()-0.5)));

    //Create a Vector3D object that will be used
to
    // control the motion of the target.
    GM01.Vector3D targetVec = new GM01.Vector3D(
        new GM01.ColMatrix3D(
            100*(random.nextDouble()-0.5),
            100*(random.nextDouble()-0.5),
            100*
(random.nextDouble()-0.5))).scale(0.3);

    //Create a reference point to mark the
origin.

```

```

    GM01.Point3D zeroPoint =
        new GM01.Point3D(new
GM01.ColMatrix3D(0,0,0));

    //Declare a variable that will be used to
control
    // the update frequency of the target
vector.
    int animationLoopCounter = 0;

    //This is the animation loop. The value of
animate
    // is set to true by the Start button and is
set to
    // false by the Stop button. Setting it to
false
    // will cause the run method to terminate
and stop
    // the animation.
    while(animate){
        animationLoopCounter++;

        //Try to keep the points from colliding
with one
        // another.
        for(int row = 0;row < points.size();row++)
        {
            for(int col = 0;col <
points.size();col++){
                GM01.Point3D refPoint =
points.get(row);
                GM01.Point3D testPoint =
points.get(col);

                if(col != row){
                    //Get the distance of the testPoint
from the

```

```

        // refPoint.
        tempVector = testPoint.

getDisplacementVector(refPoint);

        if(tempVector.getLength() < 25){
            //Modify testPoint to move it away
from
            // the refPoint and save the
modified
            // point.
            tempVector = tempVector.negate();
            tempPoint =
testPoint.addVectorToPoint(

tempVector.scale(0.1));
            points.set(col,tempPoint.clone());
        }else{
            //Do nothing.
        }//end else
    }//end if col != row
} //end loop on col
} //end loop on row

//Move all of the points toward the target
but try
// to keep them separated from the target.
for(int cnt = 0;cnt < points.size();cnt++)
{
    //Get the next point.
    tempPoint = points.get(cnt);
    //Find the distance from this point to
the
    // target.
    tempVector =

tempPoint.getDisplacementVector(target);

```

```

        if(tempVector.getLength() < 10){
            //Modify the point to move it away
from the
            // target and save the modified point.
            tempVector = tempVector.negate();
            tempPoint =
tempPoint.addVectorToPoint(
tempVector.scale(0.1));
            //Save the modified point.
            points.set(cnt,tempPoint.clone());
        }else{
            //Modify the point to move it toward
the
            // target and save the modified point.
            tempPoint =
tempPoint.addVectorToPoint(
tempVector.scale(0.2));
            points.set(cnt,tempPoint.clone());
        }//end else

        //Save a normalized version of the
direction
        // vector for drawing later.
        vectors.set(
cnt,tempVector.normalize().scale(15.0));

    }//end for loop

    //Insert a delay.
    try{
        Thread.currentThread().sleep(166);
    }catch(Exception e){
        e.printStackTrace();
    }//end catch

```

```

-
//Erase the screen
g2D.setColor(Color.WHITE);
GM01.fillRect(g2D, -osiWidth/2, osiHeight/2,
osiWidth, osiHeight);
g2D.setColor(Color.BLACK);

//Draw the points and the vectors in the
arrayList
// objects.
for(int cnt = 0; cnt < numberPoints; cnt++){
    tempPoint = points.get(cnt);
    if(drawPoints){
        tempPoint.draw(g2D);
    }//end if

    if(drawVectors){
        tempVector = vectors.get(cnt);
        tempVector.draw(g2D, tempPoint);
    }//end if
} //end for loop

//Cause the targets movements to be
slightly less
// random by using the same vector for
several
// successive movements
if(animationLoopCounter%10 == 0){
    //Get a new vector
    targetVec = new GM01.Vector3D(
        new GM01.ColMatrix3D
            (100*(random.nextDouble()-0.5),
            100*(random.nextDouble()-0.5),
            100*
(random.nextDouble()-0.5))).scale(0.3);
    }//else use the same vector again.

```

```

        //Test to see if this displacement vector
will        // push the target outside the limit. If
so,        // modify the vector to send the target
back        // toward the origin.

if(target.getDisplacementVector(zeroPoint).
                                getLength() >
0.4*osiWidth){
    targetVec =
target.getDisplacementVector(

zeroPoint).scale(0.1);
    }//end if

    //Modify the location of the target point.
    target =
target.addVectorToPoint(targetVec);

    //Set the color to RED and draw the target
and its    // displacement vector.
    g2D.setColor(Color.RED);
    target.draw(g2D);
    targetVec.normalize().scale(15.0).draw(

g2D,target);

    g2D.setColor(Color.BLACK);

    myCanvas.repaint();
} //end while loop
} //end run
} //end inner class named Animate

```

```
//=====
=====//
```

```
}//end class GUI
```

Listing 34 . Source code for the program named StringArt04.

```
/*StringArt04.java
Copyright 2008, R.G.Baldwin
Revised 03/03/08
```

This program animates the behavior of the earlier program named StringArt03. See the comments at the beginning of that program for a description of both programs. The only significant difference in the behavior of the two programs is that this program slows the rotation process down so that the user can see it happening in real time. Of course, quite a few changes were required to convert the program from a static program to an animated program.

Tested using JDK 1.6 under WinXP.

```
*****
*****/
```

```
import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;
import java.awt.event.*;
```

```
class StringArt04{
```



```

    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
} //end controlling class StringArt04
//=====
=====//

```

```

class GUI extends JFrame implements
ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 300;
    int vSize = 470;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.

    JTextField numberPointsField;//User input field.
    JTextField loopsField;//User input field.
    int numberPoints = 6;//Can be modified by the
    user.
    int loopLim = 1;//Can be modified by the user.

    JTextField zRotationField;//User input field
    JTextField xRotationField;//User input field
    JTextField yRotationField;//User input field
    double zRotation;//Rotation around z in degrees.
    double xRotation;//Rotation around x in degrees.
    double yRotation;//Rotation around y in degrees

    JTextField xAnchorPointField;//User input field
    JTextField yAnchorPointField;//User input field
    JTextField zAnchorPointField;//User input field
    double xAnchorPoint;//Rotation anchor point.

```

```

double yAnchorPoint;//Rotation anchor point.
double zAnchorPoint;//Rotation anchor point.

GM01.Point3D anchorPoint;
// boolean animate = false;

//The following variable is used to refer to an
array
// object containing the points that define the
// vertices of a geometric object.
GM01.Point3D[] points;

//-----
-----//

GUI(){//constructor

//Set JFrame size, title, and close operation.
setSize(hSize,vSize);
setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Instantiate the user input components.
numberPointsField =new JTextField("6");
loopsField = new JTextField("1");
zRotationField = new JTextField("0");
xRotationField = new JTextField("0");
yRotationField = new JTextField("0");
xAnchorPointField = new JTextField("0");
yAnchorPointField = new JTextField("0");
zAnchorPointField = new JTextField("0");
JButton button = new JButton("Animate");

//Instantiate a JPanel that will house the
user input
// components and set its layout manager.

```

```

    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,2));

    //Add the user input components and
appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" Number
Points"));
    controlPanel.add(numberPointsField);
    controlPanel.add(new JLabel(" Number Loops"));
    controlPanel.add(loopsField);
    controlPanel.add(new JLabel(
                                " Rotate around Z
(deg)"));
    controlPanel.add(zRotationField);
    controlPanel.add(new JLabel(
                                " Rotate around X
(deg)"));
    controlPanel.add(xRotationField);
    controlPanel.add(new JLabel(
                                " Rotate around Y
(deg)"));
    controlPanel.add(yRotationField);
    controlPanel.add(new JLabel(" X anchor
point"));
    controlPanel.add(xAnchorPointField);
    controlPanel.add(new JLabel(" Y anchor
point"));
    controlPanel.add(yAnchorPointField);
    controlPanel.add(new JLabel(" Z anchor
point"));
    controlPanel.add(zAnchorPointField);
    controlPanel.add(button);

    //Add the control panel to the SOUTH position
in the
    // JFrame.

```

```
        this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

        //Create a new drawing canvas and add it to
the
        // CENTER of the JFrame above the control
panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER,myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center of the
off-screen
        // image.
```

```

GM01.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

    //Register this object as an action listener
on the
    // button.
    button.addActionListener(this);

    //Cause the overridden paint method belonging
to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor
//-----
-----//

    //The purpose of this method is to draw the
appropriate
    // material onto the off-screen image.
    void drawOffScreen(Graphics2D g2D){
        //Erase the off-screen image and draw new
axes.
        setCoordinateFrame(g2D);

        //The following ColMatrix3D object must be
populated
        // with three rotation angles in degrees that
        // specify the following rotational angles in
order
        // according to the right-hand rule as applied
to
        // the indicated axis.
        // Rotate around z, in x-y plane
        // Rotate around x, in y-z plane
        // Rotate around y, in x-z plane
        GM01.ColMatrix3D angles = new
GM01.ColMatrix3D(

```

```

zRotation,xRotation,yRotation);

    //The following code causes the anchor point
to be
    // drawn and the points in the array to be
rotated.
    g2D.setColor(Color.BLACK);
    anchorPoint.draw(g2D);
    for(int cnt = 0;cnt < numberPoints;cnt++){
        points[cnt] = points[cnt].rotate(
anchorPoint,angles);
    }//end for loop

    //Implement the algorithm that draws lines
connecting
    // points on the geometric object.
    GM01.Line3D line;

    //Begin the outer loop.
    for(int loop = 1;loop <= loopLim;loop++){
        //The following variable specifies the array
        // element containing a point on which a
line will
        // start.
        int start = -1;

        //The following variable specifies the
number of
        // points that will be skipped between the
starting
        // point and the ending point for a line.
        int skip = loop;
        //The following logic causes the element
number to
        // wrap around when it reaches the end of

```

the

```
    // array.
    while(skip >= 2*numberPoints-1){
        skip -= numberPoints;
    }//end while loop

    //The following variable specifies the array
    // element containing a point on which a
line will
    // end.
    int end = start + skip;

    //Begin inner loop. This loop actually
constructs
    // the GM01.Line3D objects and causes visual
    // manifestations of those objects to be
drawn on
    // the off-screen image. Note the
requirement to
    // wrap around when the element numbers
exceed the
    // length of the array.
    for(int cnt = 0;cnt < numberPoints;cnt++){
        if(start < numberPoints-1){
            start++;
        }else{
            //Wrap around
            start -= (numberPoints-1);
        }//end else

        if(end < numberPoints-1){
            end++;
        }else{
            //Wrap around.
            end -= (numberPoints-1);
        }//end else
    }
```

```

        //Create some interesting colors.
        g2D.setColor(new
Color(cnt*255/numberPoints,

127+cnt*64/numberPoints,

255-
cnt*255/numberPoints));

        //Create a line that connects points on
the
        // geometric object.
        line = new
GM01.Line3D(points[start],points[end]);
        line.draw(g2D);
    }//end inner loop
}//end outer loop
}//end drawOffScreen
//-----
-----//

//This method is used to draw orthogonal
// 3D axes on the image that intersect at the
origin.
private void setCoordinateFrame(
        Graphics2D g2D){

    //Erase the screen
    g2D.setColor(Color.WHITE);
    GM01.fillRect(g2D, -osiWidth/2,osiHeight/2,

osiWidth,osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM01.Point3D pointA = new GM01.Point3D(
        new GM01.ColMatrix3D(-
osiWidth/2,0,0));

```



```

        GM01.Point3D pointB = new GM01.Point3D(
            new
GM01.ColMatrix3D(osiWidth/2,0,0));
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA = new GM01.Point3D(
            new GM01.ColMatrix3D(0, -
osiHeight/2,0));
        pointB = new GM01.Point3D(
            new
GM01.ColMatrix3D(0,osiHeight/2,0));
        new GM01.Line3D(pointA,pointB).draw(g2D);

        //Draw z-axis in BLUE. Make its length the
same as the
        // length of the x-axis.
        g2D.setColor(Color.BLUE);
        pointA = new GM01.Point3D(
            new GM01.ColMatrix3D(0,0, -
osiWidth/2));
        pointB = new GM01.Point3D(
            new
GM01.ColMatrix3D(0,0,osiWidth/2));
        new GM01.Line3D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method
    //-----
    -----//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){
        //Get user input values and use them to modify
several

```

```
// values that control the drawing.
numberPoints = Integer.parseInt(
numberPointsField.getText());

    loopLim =
Integer.parseInt(loopsField.getText());

    //Rotation around z in degrees.
    zRotation =

Double.parseDouble(zRotationField.getText());
    //Rotation around x in degrees.
    xRotation =

Double.parseDouble(xRotationField.getText());
    //Rotation around y in degrees
    yRotation =

Double.parseDouble(yRotationField.getText());

    //Rotation anchor point values.
    xAnchorPoint =

Double.parseDouble(xAnchorPointField.getText());
    yAnchorPoint =

Double.parseDouble(yAnchorPointField.getText());
    zAnchorPoint =

Double.parseDouble(zAnchorPointField.getText());

    //The following object contains the 3D
coordinates
    // of the point around which the rotations
will
```

```

        // take place.
        anchorPoint = new GM01.Point3D(
            new GM01.ColMatrix3D(
xAnchorPoint,yAnchorPoint,zAnchorPoint));

        //Instantiate a new array object with a length
        // that matches the new value for
numberPoints.
        points = new GM01.Point3D[numberPoints];

        //Create a set of Point3D objects that specify
        // locations on the circumference of a circle
that
        // is in the x-y plane with a radius of 50
units. Save
        // references to the Point3D objects in the
array.
        for(int cnt = 0;cnt < numberPoints;cnt++){
            points[cnt] = new GM01.Point3D(
                new GM01.ColMatrix3D(
50*Math.cos((cnt*360/numberPoints)*Math.PI/180),
50*Math.sin((cnt*360/numberPoints)*Math.PI/180),
                0.0));
        }//end for loop that creates the points

        //Spawn an animation thread
        Animate animate = new Animate();
        animate.start();

    }//end actionPerformed

//=====
====//

```

```

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the update method to eliminate the
default
    // clearing of the Canvas in order to reduce
or
    // eliminate the flashing that that is often
caused by
    // such default clearing.
    //In this case, it isn't necessary to clear
the canvas
    // because the off-screen image is cleared
each time
    // it is updated. This method will be called
when the
    // JFrame and the Canvas appear on the screen
or when
    // the repaint method is called on the Canvas
object.
    public void update(Graphics g){
        paint(g); //Call the overridden paint method.
    } //end overridden update()

    //Override the paint() method. The purpose of
the
    // paint method is to display the off-screen
image on
    // the screen. This method is called by the
update
    // method above.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    } //end overridden paint()
} //end inner class MyCanvas

```

```
//=====
===//
```

```
//This is an animation thread.
class Animate extends Thread{
    public void run(){
        //Compute incremental rotation values.
        int steps = 100;//Number of steps in the
animation.
        double zRotationInc = zRotation/steps;
        double xRotationInc = xRotation/steps;
        double yRotationInc = yRotation/steps;

        //Do the animated rotation in three distinct
stages,
        // rotating around one axis during each
stage.
        for(int axis = 0;axis < 3;axis++){
            for(int cnt = 0;cnt < steps;cnt++){
                //Select the axis about which the image
will
                // be rotated during this step..
                if(axis % 3 == 0){
                    zRotation = zRotationInc;
                    xRotation = 0;
                    yRotation = 0;
                }else if(axis % 3 == 1){
                    zRotation = 0;
                    xRotation = xRotationInc;
                    yRotation = 0;
                }else if(axis % 3 == 2){
                    zRotation = 0;
                    xRotation = 0;
                    yRotation = yRotationInc;
                }//end else

                //Draw a new off-screen image based on
```

```

        // user input values.
        drawOffScreen(g2D);
        //Copy off-screen image to canvas.
        myCanvas.repaint();

        //Sleep for ten milliseconds.
        try{
            Thread.currentThread().sleep(10);
        }catch(InterruptedException ex){
            ex.printStackTrace();
        }//end catch
    }//end for loop
} //end for loop on axis
} //end run
} //end inner class named Animate

//=====
====//

} //end class GUI

```

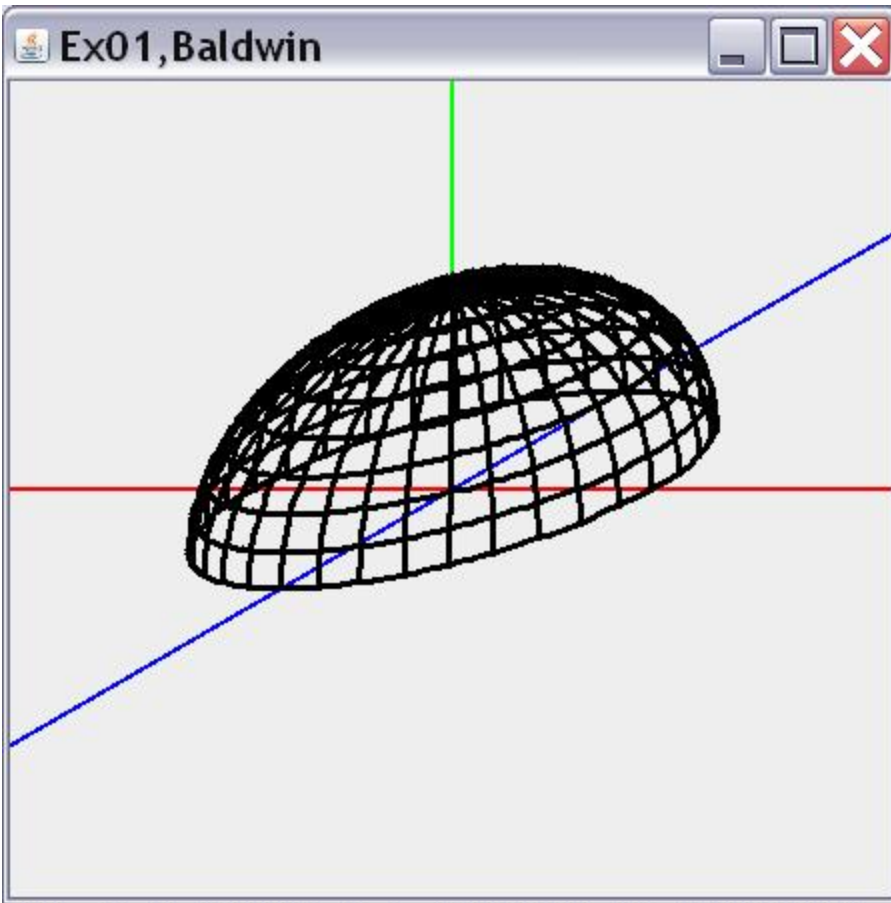
Exercises

Exercise 1

Using Java and the game-math library named **GM01** , or using a different programming environment of your choice, write a program that creates the wireframe drawing of half of a sphere protruding upward from the x-z plane as shown in [Figure 11](#). The north and south poles of the sphere lie on the y-axis. The x-z plane intersects the sphere at the equator and only the northern hemisphere is visible.

Cause your name to appear in the screen output in some manner.

Figure 11 Screen output from Exercise 1.



Exercise 2

Using Java and the game-math library named **GM01** , or using a different programming environment of your choice, write a program that demonstrates your ability to animate two or more objects and have them moving around in a 3D world.

Cause your name to appear in the screen output in some manner.

Exercise 3

Beginning with a 3D world similar to the one that you created in [Exercise 2](#) , demonstrate your ability to cause your program to recognize when two of

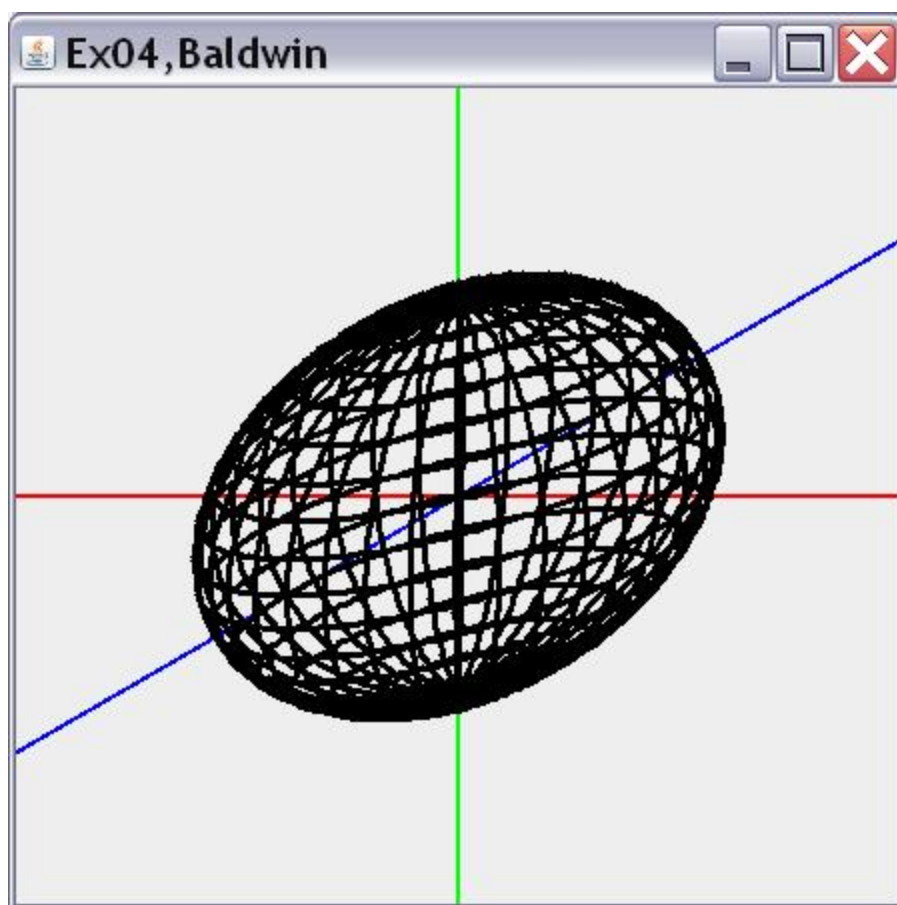
your animated objects collide and to do something that can be noticed by a human observer when the collision occurs.

Cause your name to appear in the screen output in some manner.

Exercise 4

Beginning with a program similar to the one that you wrote in [Exercise 1](#), create a wireframe drawing of a complete sphere as shown in [Figure 12](#). The north and south poles of the sphere lie on the y-axis. As before, the x-z plane intersects the sphere at the equator, but in this case, the entire sphere is visible.

Figure 12 Screen output from Exercise 4.

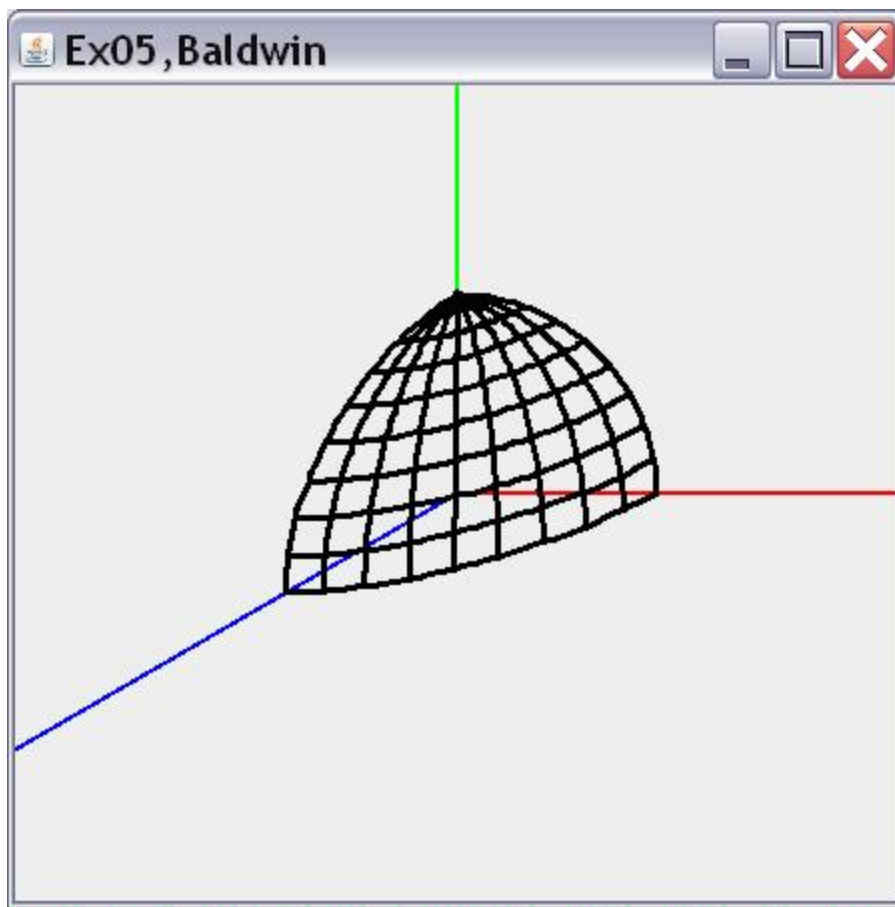


Exercise 5

Beginning with a program similar to the one that you wrote in [Exercise 1](#), create a wireframe drawing of one-eighth of a sphere as shown in [Figure 13](#). The north and south poles of the sphere lie on the y-axis. As before, the x-z plane intersects the sphere at the equator. In this case, only that portion of the sphere described by positive coordinate values is visible.

Cause your name to appear in the screen output in some manner.

Figure 13 Screen output from Exercise 5.



-end-

GAME 2302-0145: Getting Started with the Vector Dot Product
Learn the fundamentals of the vector dot product in both 2D and 3D. Learn how to update the game-math library to support various aspects of the vector dot product. Learn how to write 2D and 3D programs that use the vector dot product methods in the game-math library.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The game-math library named GM02](#)
 - [The program named DotProd2D01](#)
 - [The program named DotProd2D02](#)
 - [The program named DotProd3D01](#)
 - [The program named DotProd3D02](#)
 - [Interpreting the vector dot product](#)
 - [More than three dimensions](#)
- [Documentation for the GM02 library](#)
- [Homework assignment](#)
- [Run the programs](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listings](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)
 - [Exercise 3](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, you learned how to write your first interactive 3D game using the game-math library. You also learned how to write a Java program that simulates flocking behavior such as that exhibited by birds and fish and how to incorporate that behavior into a game. Finally, you examined three other programs that illustrate various aspects of both 2D and 3D animation using the game-math library.

What you will learn

This module is the first part of a two-part mini-series on the *vector dot product*. By the time you finish both parts, you will have learned

- the fundamentals of the vector dot product in both 2D and 3D,
- how to update the game-math library to support various aspects of the vector dot product, and
- how to write 2D and 3D programs that use the vector dot product for various applications such as the back-face culling procedure that was used to convert the image in [Figure 1](#) to the image in [Figure 2](#).

Figure 1 A 3D image before back-face culling.

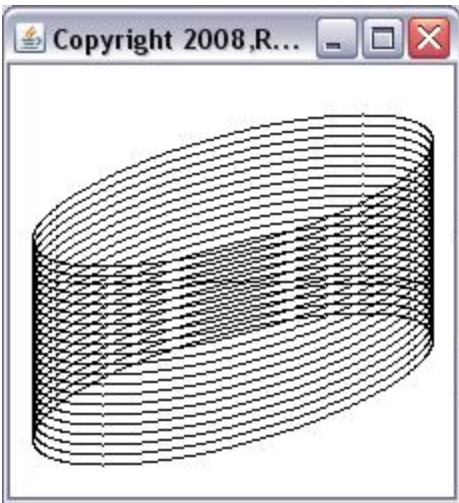
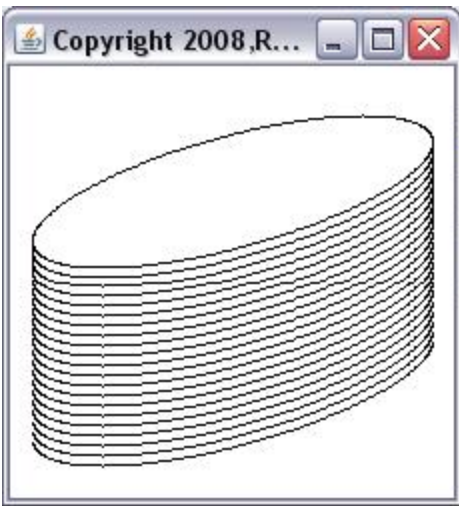


Figure 2 The 3D image after back-face culling.



Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). A 3D image before back-face culling.

- [Figure 2](#). The 3D image after back-face culling.
- [Figure 3](#). Two vectors with their tails at the origin, program DotProd2D02.
- [Figure 4](#). Dot product of two vectors with the same orientation in 2D.
- [Figure 5](#). Dot product of two vectors with the same orientation in 3D.
- [Figure 6](#). Dot product of a 3D vector with an identical vector.
- [Figure 7](#). Dot product of vectors with opposite orientations.
- [Figure 8](#). The dot product of perpendicular vectors in 2D.
- [Figure 9](#). A general formulation of 2D vector perpendicularity.
- [Figure 10](#). Another interesting 2D case of perpendicular vectors.
- [Figure 11](#). A general formulation of 3D vector perpendicularity.
- [Figure 12](#). A pair of perpendicular 3D vectors.
- [Figure 13](#). Another pair of perpendicular 3D vectors.
- [Figure 14](#). GUI for the program named DotProd2D01.
- [Figure 15](#). A screen shot of the output from the program named DotProd3D01.
- [Figure 16](#). Six (magenta) vectors that are perpendicular to a given (black) vector.
- [Figure 17](#). Output from Exercise 1.
- [Figure 18](#). Output from Exercise 2.
- [Figure 19](#). Output from Exercise 3.

Listings

- [Listing 1](#). Source code for the method named GM02.ColMatrix3D.dot.
- [Listing 2](#). Source code for the method named GM02.Vector3D.dot.
- [Listing 3](#). Source code for the method named GM02.Vector3D.angle.
- [Listing 4](#). The actionPerformed method in the program named DotProd2D01.
- [Listing 5](#). Format the dot product value for display in the GUI.
- [Listing 6](#). Beginning of the actionPerformed method in the program named DotProd2D02.
- [Listing 7](#). Compute the dot product and the angle between the two vectors.
- [Listing 8](#). Source code for the game-math library named GM02.

- [Listing 9](#). Source code for the program named DotProd2D01.
- [Listing 10](#). Source code for the program named DotProd2D02.
- [Listing 11](#). Source code for the program named DotProd3D01.
- [Listing 12](#). Source code for the program named DotProd3D02.

Preview

The homework assignment for this module was to study the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* .

I won't repeat everything that Dr. Kjell has to say. However, there are a few points that I will summarize in this section.

Basic definition of the vector dot product

The vector dot product is a special way to multiply two vectors to produce a real result. A description of the vector dot product follows.

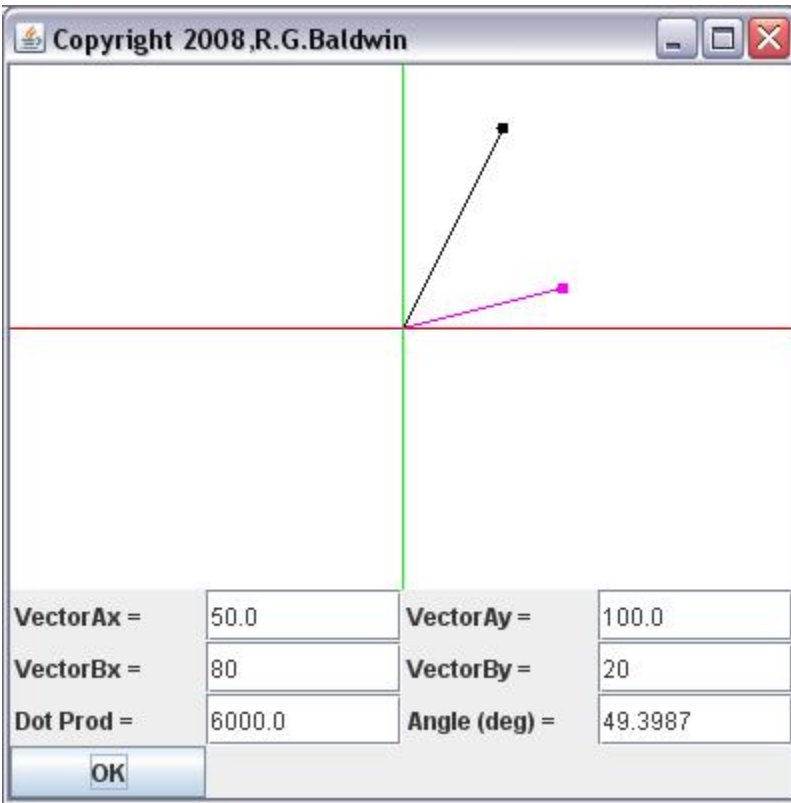
Note: The vector dot product:

The vector dot product of two vectors is the product of the lengths of the vectors multiplied by the cosine of the angle between them

By *the angle between them* , I mean the angle that would be formed if you were to draw the two vectors with their tails in the same location.

For example, [Figure 3](#) shows a black vector and a magenta vector drawn with their tails at the origin. Eyeballing the picture suggests that the angle between the two vectors is forty or fifty degrees.

Figure 3 Two vectors with their tails at the origin, program DotProd2D02.



Can do more than eyeball

Fortunately, we can do more than eyeball the angle between two vectors. [Figure 3](#) shows the screen output produced by the program named **DotProd2D02** that I will explain in this module. **DotProd2D02** is a 2D program. I will also explain a 3D version named **DotProd3D02** in this module as well.

In [Figure 3](#), the top four user input fields allow the user to enter the x and y coordinate values of two vectors according to the labels that identify those fields. When the user clicks the OK button, the first vector is drawn in black with its tail at the origin and the second vector is drawn in magenta with its tail at the origin. The dot product of the two vectors is computed and displayed in the bottom left text field, and the angle between the two vectors is computed and displayed in the bottom right text field.

Don't need to know the angle between the vectors

Upon seeing the description of the dot product given [above](#), you may reasonably be concerned about needing to know the angle between the vectors before you can compute the dot product. Fortunately, as you will see [later](#), it is possible to compute the dot product of two vectors without knowing the angle. In fact, being able to compute the dot product is one way to determine the angle between two vectors.

As you can see, the value of the dot product of the two vectors shown in [Figure 3](#) is 6000 and the angle between the vectors is 49.3987 degrees. You will learn how those values were computed shortly.

Major properties of the dot product

Here are some of the major properties of the dot product of two vectors:

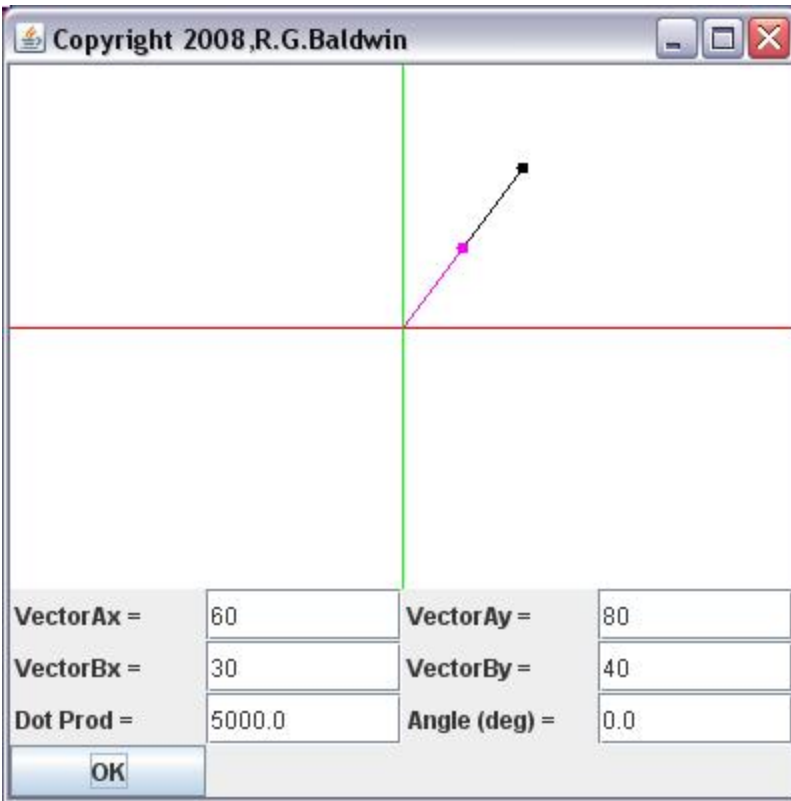
1. The dot product of two vectors with the same orientation is the product of their lengths.
2. The length of a vector is the square root of the dot product of the vector with itself.
3. The dot product of two vectors having opposite orientations is the negative of the product of their lengths.
4. The dot product of perpendicular vectors is zero.
5. The angle between two vectors is the same as the angle between normalized versions of the vectors, which is equal to the arc cosine of the dot product of the normalized vectors.

As you will see later, these properties apply to both 2D and 3D vectors. In many cases, they also apply to vectors having more than three dimensions as well.

Dot product of two vectors with the same orientation in 2D

[Figure 4](#) illustrates the first property in the above list: *The dot product of two vectors with the same orientation is the product of their lengths.*

Figure 4 Dot product of two vectors with the same orientation in 2D.



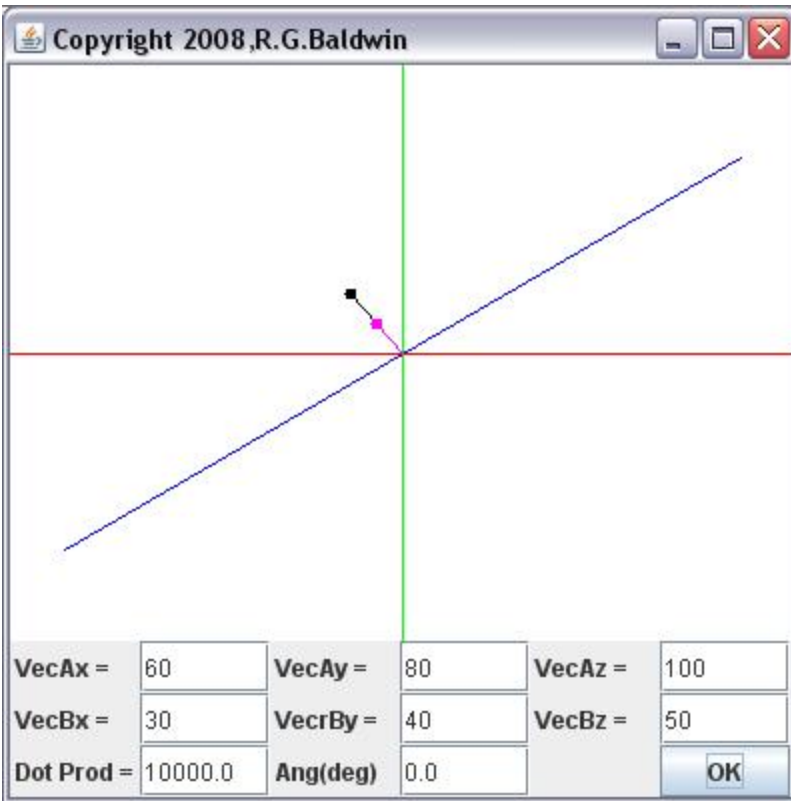
You may recall from some earlier experience that when a right triangle has sides with lengths of 30 and 40, the length of the hypotenuse is 50. That is the case for the magenta vector shown in [Figure 4](#). Similarly, when the sides of the triangle are 60 and 80, the length of the hypotenuse is 100, as is the case for the black vector in [Figure 4](#).

From the property given above, we know that the dot product of the black and magenta vectors shown in [Figure 4](#) is 5000, which agrees with the value shown in the **Dot Prod** output field in [Figure 4](#).

Dot product of two vectors with the same orientation in 3D

[Figure 5](#) shows the dot product of two vectors with the same orientation in 3D. The image in [Figure 5](#) was produced by the program named **DotProd3D02**.

Figure 5 Dot product of two vectors with the same orientation in 3D.



Manually calculate the value of the dot product

You may need to get your calculator out to manually compute the lengths of the two vectors in [Figure 5](#). Computing the lengths as the square root of the sum of the squares of the three components of each vector gives me the following lengths:

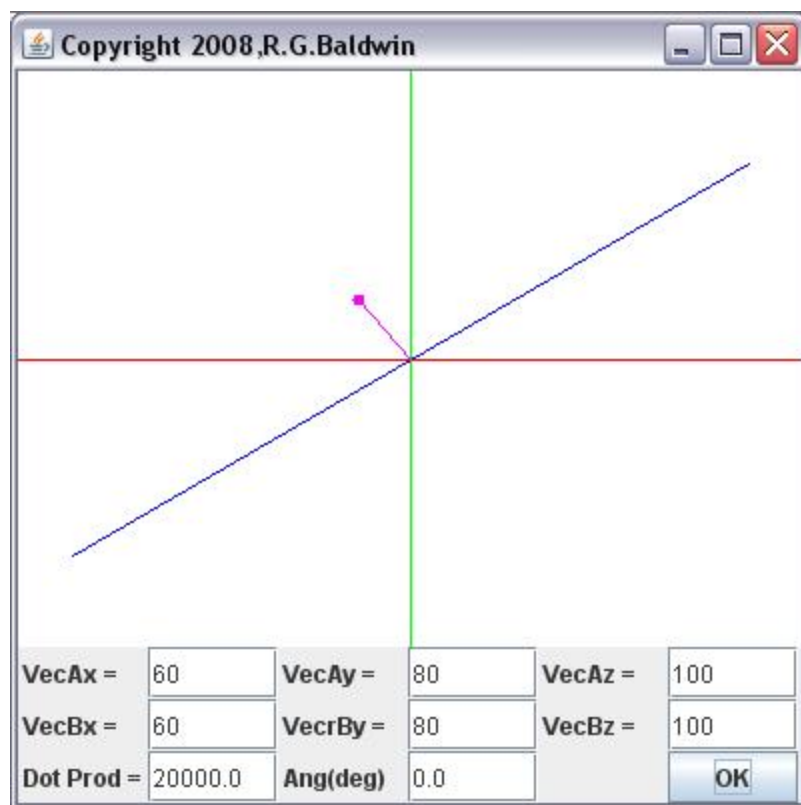
- Black length = 141.42
- Magenta length = 70.71

Rounding the product of the two lengths gives the dot product value of 10000, which matches the value shown in the bottom left output field in [Figure 5](#).

The length of a vector

[Figure 6](#) illustrates the second property in the above [list](#): *The length of a vector is the square root of the dot product of the vector with itself.*

Figure 6 Dot product of a 3D vector with an identical vector.



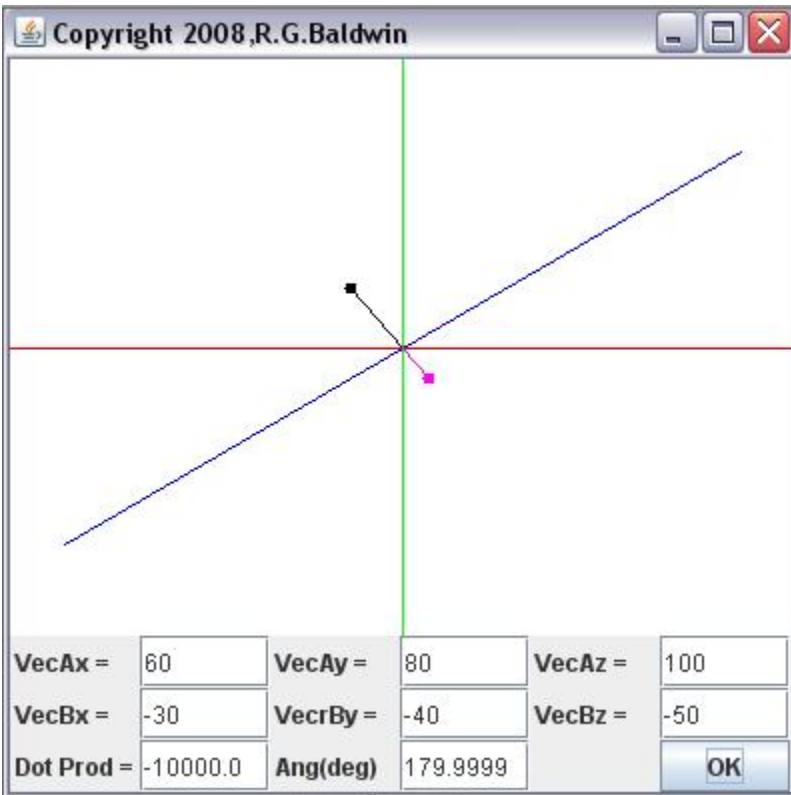
[Figure 6](#) displays two vectors having the same coordinate values as the black vector in [Figure 5](#). (The black vector is hidden by the magenta vector in [Figure 6](#).) Because these two vectors have identical coordinate values, the dot product of these two vectors is the same as the dot product of either vector with itself.

We concluded earlier that the length of each of these vectors is 141.42. This length is the square root of the dot product of the vector with itself. Squaring and rounding this length gives a dot product value of 20000, which matches the value shown in the bottom left output field in [Figure 6](#).

Dot product of vectors with opposite orientations

[Figure 7](#) illustrates the third property of the dot product given [above](#): The dot product of two vectors having opposite orientations is the negative of the product of their lengths.

Figure 7 Dot product of vectors with opposite orientations.



The two vectors shown in [Figure 7](#) have the same absolute coordinates as the two vectors shown in [Figure 5](#). However, the algebraic signs of the coordinates of the magenta vector in [Figure 7](#) were reversed relative to [Figure 4](#), causing the magenta vector to point in the opposite direction from the black vector. (Note that the angle between the two vectors, as reported by the program is zero degrees in [Figure 5](#) and is 180 degrees in [Figure 7](#).)

The point here is that the dot product of the two vectors in [Figure 7](#) is the negative of the dot product of the two vectors in [Figure 5](#). This property will be used in another program in the second part of this two-part miniseries to achieve the back-face culling shown in [Figure 1](#).

The computational simplicity of the vector dot product

If you have studied the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* you have learned that the dot product of two vectors can be computed as the sum of products of the corresponding x, y, and z components of the two vectors. In particular, **in the 2D case**, the dot product is given by:

$$\text{2D dot product} = x1*x2 + y1*y2$$

Similarly, **in the 3D case** , the dot product is given by:

$$\text{3D dot product} = x1*x2 + y1*y2 + z1*z2$$

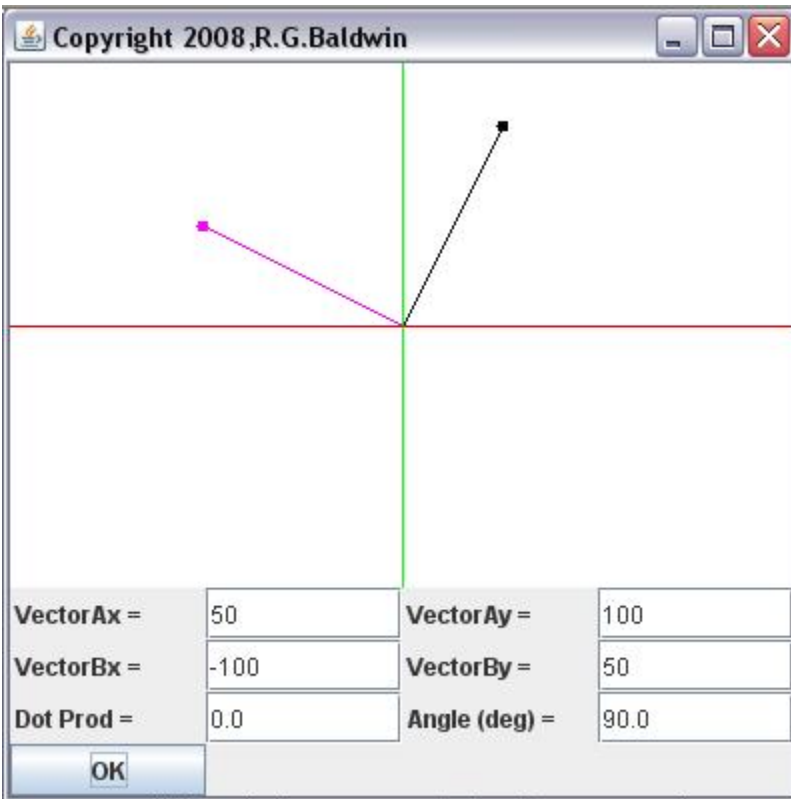
Note that these two formulations don't require the program to know anything about the angle between the two vectors, as is the case in the [earlier](#) definition.

The dot product of perpendicular vectors in 2D

The dot product has many uses in game programming, not the least of which is determining if two vectors are perpendicular.

[Figure 8](#) illustrates the fourth property in the above [list](#): *The dot product of perpendicular vectors is zero* . This is an extremely important property in that it allows game programs to easily determine if two vectors are perpendicular. I will begin with a 2D discussion because the topic of perpendicularity of vectors is **less complicated in 2D than in 3D** .

Figure 8 The dot product of perpendicular vectors in 2D.



By eyeballing [Figure 8](#), you can see that the magenta vector is probably perpendicular to the black vector. Looking at the output fields at the bottom left and the bottom right in [Figure 8](#), you will see that the dot product of the two vectors is zero and the angle between the two vectors is 90 degrees.

Restating the perpendicularity property

Most of us learned in our earlier mathematics classes that the angle between perpendicular lines is 90 degrees. Therefore, the angle between perpendicular vectors must be 90 degrees. (See [a definition of perpendicularity](#).)

Most of us also learned in our trigonometry classes that the cosine of 90 degrees is 0.0. Since, [by definition](#), *the value of the dot product of two vectors is the product of the lengths of the vectors multiplied by the cosine of the angle between them*, and the angle must be 90 degrees for two vectors to be perpendicular, then the dot product of perpendicular vectors must be zero as stated by the fourth property in the above [list of properties](#).

An infinite number of perpendicular vectors

By observing [Figure 8](#), it shouldn't be too much of a stretch for you to recognize that there are an infinite number of different vectors that could replace the magenta vector in [Figure 8](#) and be perpendicular to the black vector. However, since we are discussing the 2D case here, all of those vectors must lie in the same plane and must have the same orientation (*or the reverse orientation*) as the magenta vector. In other words, all of the vectors in the infinite set of vectors that are perpendicular to the black vector must lie on the line defined by the magenta vector, pointing in either the same direction or in the opposite direction. However, those vectors can be any length and still lie on that same line.

A general formulation of 2D vector perpendicularity

By performing some algebraic manipulations on the [earlier](#) 2D formulation of the dot product, we can formulate the equations shown in [Figure 9](#) that define the infinite set of perpendicular vectors described above.

Figure 9 . A general formulation of 2D vector perpendicularity.

$$\text{dot product} = x_1 * x_2 + y_1 * y_2$$

If the two vectors are perpendicular:

$$x_1 * x_2 + y_1 * y_2 = 0.0$$

$$x_1 * x_2 = -y_1 * y_2$$

$$x_2 = -y_1 * y_2 / x_1$$

As you can see from [Figure 9](#), we can assume any values for y_1 , y_2 , and x_1 and compute a value for x_2 that will cause the two vectors to be

perpendicular.

A very interesting case

One very interesting 2D case is the case shown in [Figure 8](#). In this case, I initially specified one of the vectors to be given by the coordinate values (50,100). Then I assumed that y_2 is equal to x_1 and computed the value for x_2 . The result is that the required value of x_2 is the negative of the value of y_1 .

Thus, in the 2D case, we can easily define two vectors that are perpendicular by

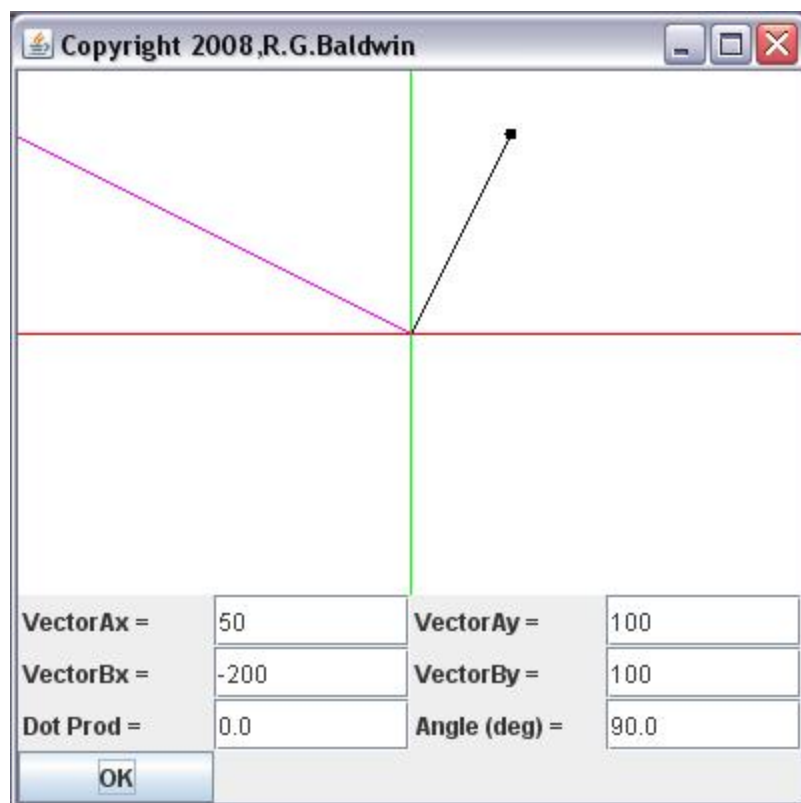
- swapping the coordinate values between two vectors and
- negating one of the coordinate values in the second vector

The actual direction that the second vector points will depend on which value you negate in the second vector.

Another interesting 2D case of perpendicular vectors

Another interesting 2D case is shown in [Figure 10](#).

Figure 10 Another interesting 2D case of perpendicular vectors.



In [Figure 10](#), I assumed the same coordinate values for the black vector as in [Figure 8](#). Then I assumed that the values of the y-coordinates for both vectors are the same. Using those values along with the equation in [Figure 8](#), I manually computed a required value of -200 for the x-coordinate of the magenta vector. I entered that value into the field labeled **VectorBx** = in [Figure 10](#) and clicked the **OK** button.

And the result was...

You can see that the value of the dot product of the two vectors in [Figure 10](#) is 0.0, and the angle between the two vectors is 90 degrees. Therefore, although the magenta vector in [Figure 10](#) is much longer than the magenta vector in [Figure 8](#), the magenta vector in [Figure 10](#) is still perpendicular to the black vector. Thus, [Figure 8](#) and [Figure 10](#) show two of the infinite number of magenta vectors that are perpendicular to the black vector in those images.

The dot product of perpendicular vectors in 3D

As I mentioned earlier, the topic of perpendicularity in 3D is [more complicated](#) than is the case in 2D. As is the case in 2D, there are an infinite number of vectors that are perpendicular to a given vector in 3D. In 2D, the infinite set of perpendicular vectors must have different lengths taken in pairs, and the vectors in each pair must point in opposite directions.

An infinite number of perpendicular vectors having the same length

However, in 3D there are an infinite number of vectors having the same length that are perpendicular to a given vector. All of the perpendicular vectors having the same length must point in different directions and they must all lie in a plane that is perpendicular to the given vector.

Perpendicular vectors having different lengths may point in the same or in different directions but they also must lie in a plane that is perpendicular to the given vector.

A wagon-wheel analogy

Kjell explains the situation of an infinite set of 3D vectors that are perpendicular to a given vector by describing an old-fashioned wagon wheel with spokes that emanate directly from the hub and extend to the rim of the wheel. The hub surrounds the axle and each of the spokes is perpendicular to the axle. Depending on the thickness of the spokes, a large (*but probably not infinite*) number of spokes can be used in the construction of the wagon wheel.

Another wheel at the other end of the axle

In this case, the wagon wheel lies in a plane that is perpendicular to the axle. There is normally another wheel at the other end of the axle. Assuming that the axle is straight, the second wheel is in a different plane but that plane is also perpendicular to the axle. Thus, the spokes in the second wheel are also perpendicular to the axle.

If there were two identical wheels at each end of the axle for a total of four wheels (*the predecessor to the modern 18-wheel tractor trailer*), the spokes in all four of the wheels would be perpendicular to the axle. Again, the

point is that there are an infinite number of vectors that are perpendicular to a given vector in 3D.

A general formulation of 3D vector perpendicularity

By performing some algebraic manipulations on the [earlier](#) 3D formulation of the dot product, we can develop the equations shown in [Figure 11](#) that describe an infinite set of vectors that are perpendicular to a given vector.

Figure 11 . A general formulation of 3D vector perpendicularity.

$$\text{dot product} = x1*x2 + y1*y2 + z1*z2$$

If the two vectors are perpendicular:

$$x1*x2 + y1*y2 + z1*z2 = 0.0$$

$$x1*x2 = -(y1*y2 + z1*z2)$$

$$x2 = -(y1*y2 + z1*z2)/x1$$

or

$$y2 = -(x1*x2 + z1*z2)/y1$$

or

$$z2 = -(x1*x2 + y1*y2)/z1$$

(Although I didn't do so in [Figure 11](#), I could have written three more equations that could be used to solve for x1, y1, and z1 if the given vector is

notated as x_2 , y_2 , and z_2 .)

No simple case

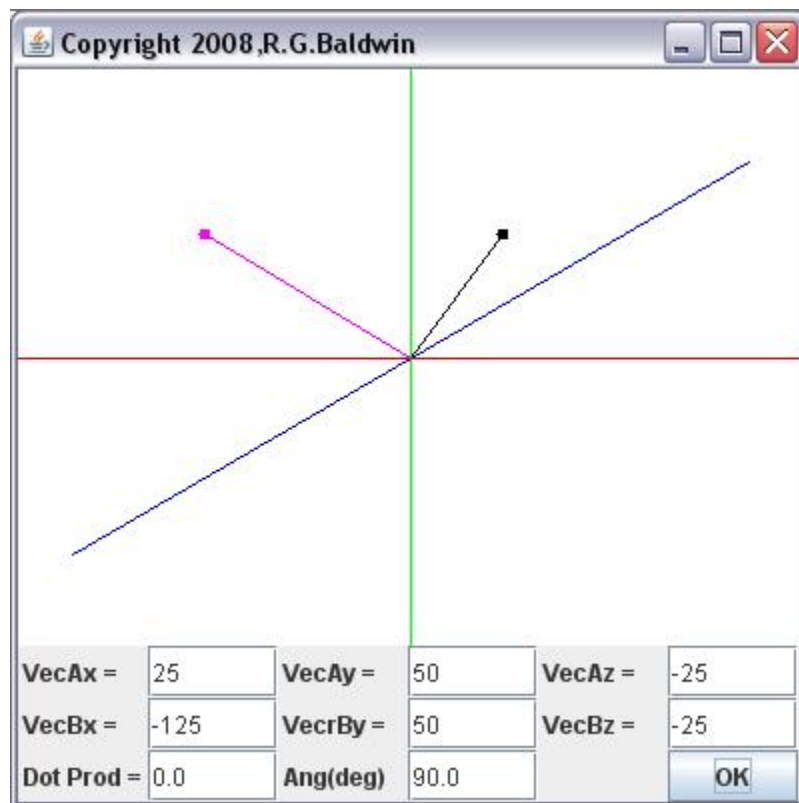
Unlike with 2D vectors, *(to my knowledge)*, there is no case that simply allows you to swap coordinate values and change the sign on one of them to produce a vector that is perpendicular to another vector. However, given the equations in [Figure 11](#), and given values for x_1 , y_1 , and z_1 , we can assume values for y_2 and z_2 and determine the value for x_2 that will cause the two vectors to be perpendicular.

While this is a fairly tedious computation with a hand calculator, it is very easy to write Java code to perform the computation. Therefore, given a 3D vector, it is fairly easy to write Java code that will compute an infinite number of vectors that are perpendicular to the given vector.

A pair of perpendicular 3D vectors

[Figure 12](#) and [Figure 13](#) each show a magenta 3D vector that is part of the infinite set of vectors that are all perpendicular to the black 3D vector. In [Figure 12](#), y_1 was assumed to be equal to y_2 , and z_1 was assumed to be equal to z_2 . For an x_1 value of 25, a value of -125 was required to cause the magenta vector to be perpendicular to the black vector.

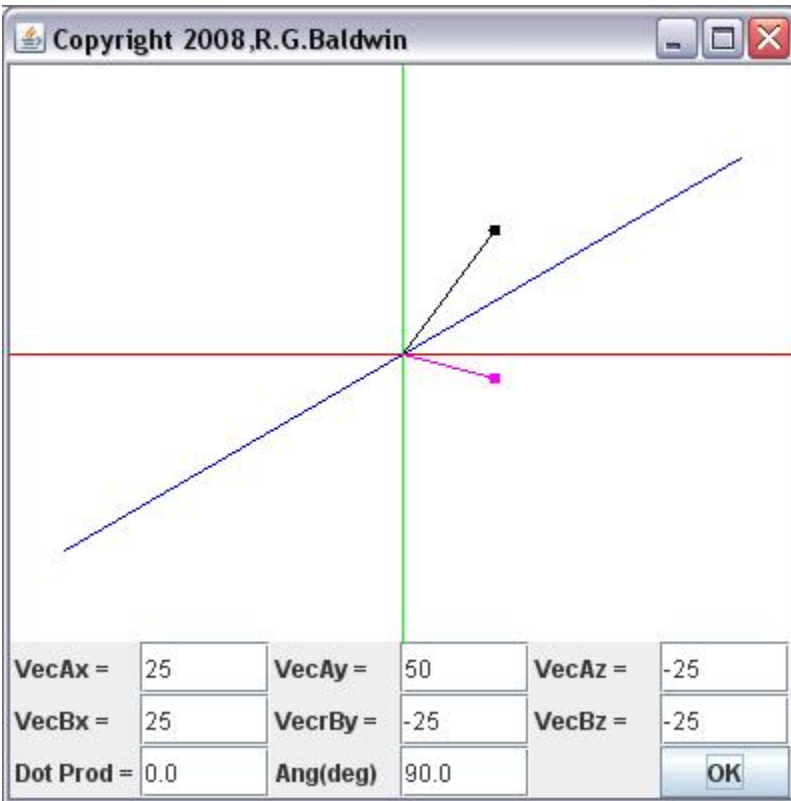
Figure 12 A pair of perpendicular 3D vectors.



Another pair of perpendicular 3D vectors

In [Figure 13](#), x_1 was assumed to be equal to x_2 , and z_1 was assumed to be equal to z_2 .

Figure 13 Another pair of perpendicular 3D vectors.



For a y1 value of 50, a y2 value of - 25 was required to cause the magenta vector to be perpendicular to the black vector.

The black vector didn't change

Note that the black vector is the same in [Figure 12](#) and [Figure 13](#). However, the magenta vectors are different in [Figure 12](#) and [Figure 13](#). They represent just two of the infinite set of vectors that are perpendicular to the black vector. *(They are two of the vectors in the infinite set of perpendicular vectors that are relatively easy to compute using program code.)*

Computing the angle between two vectors

The fifth property in the previous [list](#) reads: *The angle between two vectors is the same as the angle between normalized versions of the vectors, which is equal to the arc cosine of the dot product of the normalized vectors.*

In other words, given two vectors, we can compute the angle between the two vectors by first normalizing each vector and then computing the dot product of the two normalized vectors.

(By normalizing, I mean to change the coordinate values such that the direction of the vector remains the same but the length of the vector is converted to 1.0.)

The dot product is equal to the cosine of the angle. The actual angle can be obtained by using one of the methods of the **Math** class to compute the arc cosine of the value of the dot product.

Used to compute the output angle in the programs

This is the procedure that is used by the programs in this module to compute and display the angle between two vectors as illustrated by the contents of the output fields labeled **Ang(deg)** or **Angle (deg)** = in many of the images in this module.

I will have more to say about this topic as I explain the code in the upgraded game-math library named **GM02** as well as the following four programs that demonstrate the use of the upgraded game-math library:

- DotProd2D01
- DotProd2D02
- DotProd3D01
- DotProd3D02

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

The game-math library named GM02

In this section, I will present and explain an updated version of the game-math library named **GM02** .

This game-math library is an update to the game-math library named **GM01** . The main purpose of this update was to add *vector dot product* and related capabilities, such as the computation of the angle between two vectors to the library.

The following methods are new instance methods of the indicated static top-level classes belonging to the class named **GM02** .

- **GM02.ColMatrix2D.dot** - computes dot product of two ColMatrix2D objects.
- **GM02.Vector2D.dot** - computes dot product of two Vector2D objects.
- **GM02.Vector2D.angle** - computes angle between two Vector2D objects.
- **GM02.ColMatrix3D.dot** - computes dot product of two ColMatrix3D objects
- **GM02.Vector3D.dot** - computes dot product of two Vector3D objects.
- **GM02.Vector3D.angle** - computes angle between two Vector3D objects.

Will only explain the new 3D methods

I have explained much of the code in the game-math library in previous modules, and I won't repeat those explanations here. Rather, I will explain only the new 3D code in this module. Once you understand the new 3D code, you should have no difficulty understanding the new 2D code.

You can view a complete listing of the updated game-math library in [Listing 8](#) near the end of the module.

Source code for the method named GM02.ColMatrix3D.dot

[Listing 1](#) shows the source code for the new instance method named **GM02.ColMatrix3D.dot** . If you have studied the Kjell tutorials, you have learned that the dot product can be applied not only to two vectors, but can also be applied to the column matrix objects that are used to represent the

vectors. [Listing 1](#) computes the dot product of two **ColMatrix3D** objects and returns the result as type **double**.

Listing 1 . Source code for the method named GM02.ColMatrix3D.dot.

```
public double dot(GM02.ColMatrix3D matrix)
{
    return getData(0) * matrix.getData(0)
        + getData(1) * matrix.getData(1)
        + getData(2) * matrix.getData(2);
} //end dot
```

This is one of those cases where it is very easy to write the code once you understand the code that you need to write. [Listing 1](#) implements the equation for the 3D dot product that I provided [earlier](#) and returns the result of the computation as type **double** .

Source code for the method named GM02.Vector3D.dot

[Listing 2](#) shows the source code for the method named **GM02.Vector3D.dot** . This method computes the dot product of two **Vector3D** objects and returns the result as type **double** .

Listing 2 . Source code for the method named GM02.Vector3D.dot.

```
public double dot(GM02.Vector3D vec){
    GM02.ColMatrix3D matrixA =
getColMatrix();
    GM02.ColMatrix3D matrixB =
vec.getColMatrix();
    return matrixA.dot(matrixB);
} //end dot
```

Once again, the code is straightforward. All **Vector3D** objects instantiated from classes in the game-math library are represented by objects of the **ColMatrix3D** class. [Listing 2](#) gets a reference to the two **ColMatrix3D** objects that represent the two vectors for which the dot product is needed. Then it calls the **GM02.ColMatrix3D.dot** method shown earlier in [Listing 1](#) to get and return the value of the dot product of the two vectors.

Source code for the method named GM02.Vector3D.angle

[Listing 3](#) shows the source code for the method named **GM02.Vector3D.angle** . This method computes and returns the angle between two **Vector3D** objects. The angle is returned in degrees as type **double** .

Listing 3 . Source code for the method named GM02.Vector3D.angle.

**Listing 3 . Source code for the method named
GM02.Vector3D.angle.**

```
public double angle(GM02.Vector3D vec){
    GM02.Vector3D normA = normalize();
    GM02.Vector3D normB = vec.normalize();
    double normDotProd = normA.dot(normB);
    return
    Math.toDegrees(Math.acos(normDotProd));
} //end angle
```

You need to understand trigonometry here

If you understand trigonometry, you will find the code in [Listing 3](#) straightforward. If not, simply take my word for it that the method shown in [Listing 3](#) behaves as described above.

The method begins by calling the **normalize** method on each of the two vectors for which the angle between the vectors is needed. (*See the definition of the **normalize** method in [Listing 8](#).*)

Then [Listing 3](#) computes the dot product of the two normalized vectors. The value of the dot product is the cosine of the angle between the two original vectors.

After that, [Listing 3](#) calls the **acos** method of the **Math** class to get the *arc (inverse) cosine* (see [Inverse Cosine](#)) of the dot product value. The **acos** method returns the angle in radians.

Finally, [Listing 3](#) calls the **toDegrees** method of the **Math** class to convert the angle from radians to degrees and to return the angle in degrees as type **double** .

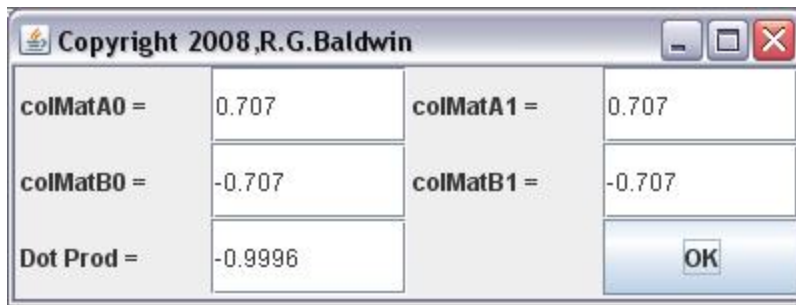
That completes the discussion of the updates to the game-math library resulting in the new library named **GM02** .

The program named DotProd2D01

To understand this program, you need to understand the material in the Kjell tutorial through *Chapter 8 - Length, Orthogonality, and the Column Matrix Dot product*.

The purpose of this program is simply to confirm proper operation of the **GM02.ColMatrix2D.dot** method. The program output is shown in [Figure 14](#).

Figure 14 GUI for the program named DotProd2D01.



A graphical user interface

The program creates a GUI that allows the user to enter the first and second values for each of a pair of **ColMatrix2D** objects into four text fields. The GUI also provides a button labeled **OK**. When the user clicks the **OK** button, the dot product of the two **ColMatrix2D** objects is computed. The resulting value is formatted to four decimal digits and displayed in a text field in the lower left of the GUI.

Similar to previous programs

Much of the code in this program is similar to code that I have explained in earlier modules, so I won't repeat those explanations here. I will explain only the code contained in the **actionPerformed** method that is new to this module. A complete listing of this program is shown in [Listing 9](#).

The actionPerformed method in the program named DotProd2D01

The beginning of the **actionPerformed** method is shown in [Listing 4](#). This method is called to respond to a click on the **OK** button shown in [Figure 14](#).

Listing 4 . The actionPerformed method in the program named DotProd2D01.

```
public void actionPerformed(ActionEvent e){
    //Create two ColMatrix2D objects.
    GM02.ColMatrix2D matrixA = new
GM02.ColMatrix2D(
        Double.parseDouble(colMatA0.getText()),
        Double.parseDouble(colMatA1.getText()));

    GM02.ColMatrix2D matrixB = new
GM02.ColMatrix2D(
        Double.parseDouble(colMatB0.getText()),
        Double.parseDouble(colMatB1.getText()));

    //Compute the dot product.
    double dotProd = matrixA.dot(matrixB);
}
```

[Listing 4](#) begins by instantiating a pair of **GM02.ColMatrix2D** objects using data provided by the user in the top four fields shown in [Figure 14](#).

Then [Listing 4](#) calls the **dot** method on the object referred to by **matrixA** , passing the object referred to by **matrixB** as a parameter. The **dot** method computes the dot product of the two column matrix objects, returning the result as type **double** .

Format the dot product value for display in the GUI

In some cases, the format of the returned value is not very suitable for display in the lower-left field in [Figure 14](#). For example, if the value is very small, it is returned in an exponential notation requiring a large number of digits for display. Similarly, sometimes the dot-product value is returned in a format something like 0.33333333 requiring a large number of digits to display.

[Listing 5](#) formats the dot product value to make it suitable for display in the text field in [Figure 14](#). *(There may be an easier way to do this, but I didn't want to take the time to figure out what it is.)*

Listing 5 . Format the dot product value for display in the GUI.

```
//Eliminate exponential notation in the
display.
if(Math.abs(dotProd) < 0.001){
    dotProd = 0.0;
}//end if

//Convert to four decimal digits and
display.
dotProd =((int)(10000*dotProd))/10000.0;
dotProduct.setText("" + dotProd);

}//end actionPerformed
```

Eliminate exponential format and format to four decimal digits

[Listing 5](#) begins by simply setting small values that are less than 0.001 to 0.0 to eliminate the exponential format for very small, non-zero values.

Then [Listing 5](#) executes some code that formats the dot product value to four decimal digits. I will leave it as an exercise for the student to decipher how this code does what it does.

I recommend that you try it

I recommend that you plug a few values into the input fields, click the **OK** button, and use your calculator to convince yourself that the program properly implements the 2D dot product equation shown [earlier](#).

That concludes the discussion of the program named **DotProd2D01** .

The program named DotProd2D02

To understand this program, you need to understand the material in the Kjell tutorial through *Chapter 9, The Angle Between Two Vectors* .

This program allows the user to experiment with the dot product and the angle between a pair of **GM02.Vector2D** objects.

A screen shot of the output from this program is shown in [Figure 3](#). The GUI shown in [Figure 3](#) is provided to allow the user to enter four double values that define each of two **GM02.Vector2D** objects. The GUI also provides an **OK** button as well as two text fields used for display of computed results.

In addition, the GUI provides a 2D drawing area. When the user clicks the **OK** button, the program draws the two vectors, (*one in black and the other in magenta*) , on the output screen with the tail of each vector located at the origin in 2D space. The program also displays the values of the dot product of the two vectors and the angle between the two vectors in degrees.

Once again, much of the code in this program is similar to code that I have explained before. I will explain only the method named **actionPerformed** ,

for which some of the code is new to this module. A complete listing of this program is provided in [Listing 10](#).

Beginning of the actionPerformed method in the program named DotProd2D02

This method is called to respond to a click on the **OK** button in [Figure 2](#).

Listing 6 . Beginning of the actionPerformed method in the program named DotProd2D02.

```
public void actionPerformed(ActionEvent e){  
    //Erase the off-screen image and draw the  
    axes.  
    setCoordinateFrame(g2D);  
  
    //Create two ColMatrix2D objects based on  
    the user  
    // input values.  
    GM02.ColMatrix2D matrixA = new  
    GM02.ColMatrix2D(  
    Double.parseDouble(vectorAx.getText()),  
    Double.parseDouble(vectorAy.getText()));  
  
    GM02.ColMatrix2D matrixB = new  
    GM02.ColMatrix2D(  
    Double.parseDouble(vectorBx.getText()),
```


Listing 6 . Beginning of the actionPerformed method in the program named DotProd2D02.

```
Double.parseDouble(vectorBy.getText()));

    //Use the ColMatrix2D objects to create
    two Vector2D
    // objects.
    GM02.Vector2D vecA = new
    GM02.Vector2D(matrixA);
    GM02.Vector2D vecB = new
    GM02.Vector2D(matrixB);

    //Draw the two vectors with their tails at
    the origin.
    g2D.setColor(Color.BLACK);
    vecA.draw(
        g2D,new GM02.Point2D(new
    GM02.ColMatrix2D(0,0)));

    g2D.setColor(Color.MAGENTA);
    vecB.draw(
        g2D,new GM02.Point2D(new
    GM02.ColMatrix2D(0,0)));
```

[Listing 6](#) gets the four user input values that define the two vectors and draws them in black and magenta on the GUI shown in [Figure 3](#). There is nothing new in the code in [Listing 6](#).

Compute the dot product and the angle between the two vectors

[Listing 7](#) computes the dot product and the angle between the two vectors by first calling the **dot** method and then the **angle** method on the object referred to by **vecA** , passing the object referred to by **vecB** as a parameter.

Listing 7 . Compute the dot product and the angle between the two vectors.

```
//Compute the dot product of the two
vectors.
double dotProd = vecA.dot(vecB);

//Output formatting code was deleted for
brevity

//Compute the angle between the two
vectors.
double angle = vecA.angle(vecB);

//Output formatting code was deleted for
brevity

myCanvas.repaint();//Copy off-screen image
to canvas.
} //end actionPerformed
```

In both cases, the code in [Listing 7](#) formats the returned **double** values to make them appropriate for display in the bottom two text fields in [Figure 3](#). This code was deleted from [Listing 7](#) for brevity.

Confirm that the results are correct

Because this is a 2D display, it is easy to make an eyeball comparison between the drawing of the two vectors and the reported angle between the two vectors to confirm agreement. However, I recommend that you use this program to define several vectors and then use your scientific calculator to confirm that the results shown are correct.

That concludes the explanation of the program named **DotProd2D02** .

The program named DotProd3D01

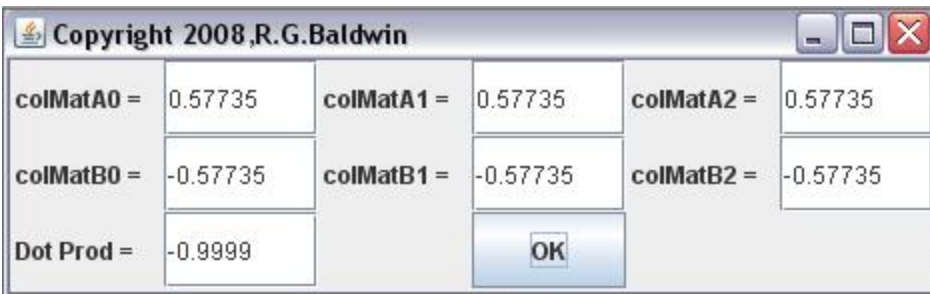
To understand this program, you need to understand the material in the Kjell tutorial through *Chapter 8 - Length, Orthogonality, and the Column Matrix Dot product* .

The purpose of this program is to confirm proper operation of the **ColMatrix3D.dot** method. The program creates a GUI that allows the user to enter three values for each of a pair of **ColMatrix3D** objects along with a button labeled **OK** . When the user clicks the **OK** button, the dot product of the two **ColMatrix3D** objects is computed and displayed.

A screen shot of the output from the program named DotProd3D01

A screen shot of the output from the program named **DotProd3D01** is shown in [Figure 15](#) . (Compare [Figure 15](#) with [Figure 14](#) .)

Figure 15 A screen shot of the output from the program named **DotProd3D01**.



Very similar to a previous program

Except for the fact that this program calls the **dot** method on an object of the **GM02.ColMatrix3D** class instead calling the **dot** method on an object of the **GM02.ColMatrix2D** class, this program is essentially the same as the program named **DotProd2D01** that I explained earlier. Therefore, you should have no trouble understanding this program without further explanation from me. A complete listing of this program is provided in [Listing 11](#) .

That concludes the explanation of the program named **DotProd3D01** .

The program named DotProd3D02

You need to understand the material in the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* to understand this program.

Program output

A screen shot of the output of this program is shown in [Figure 12](#). This program allows the user to experiment with the dot product and the angle between a pair of **GM02.Vector3D** objects. A GUI is provided that allows the user to enter six **double** values that define each of two **GM02.Vector3D** objects. The GUI also provides an **OK** button as well as two text fields used for display of the computed results.

In addition, the GUI provides a 3D drawing area. When the user clicks the **OK** button, the program draws the two vectors, (*one in black and the other in magenta*) , on the output screen with the tail of each vector located at the origin in 3D space. The program also displays the values of the dot product of the two vectors and the angle between the two vectors in degrees. (*Compare the output of this 3D program in [Figure 12](#) with the output from the 2D program in [Figure 3](#).*)

Program code

A complete listing of this program is provided in [Listing 12](#). Almost all of the new and interesting code in this program is in the method named **actionPerformed** .

Very similar to a previous program

If you compare the method named **actionPerformed** in [Listing 12](#) with the **actionPerformed** method for the program named **DotProd2D02** in [Listing 10](#), you will see that they are very similar. One calls 2D methods in the game-math library while the other calls 3D methods in the same library. Therefore, you should have no difficulty understanding this program without further explanation from me.

That concludes the explanation of the program named **DotProd3D02** .

Interpreting the vector dot product

In effect, the dot product of two vectors provides a measure of the extent to which the two vectors have the same orientation. If the two vectors are parallel, the dot product of the two vectors has a maximum value of 1.0 multiplied by the product of the lengths of the two vectors. Normalizing the vectors before computing the dot product will eliminate the effect of the vector lengths and will cause the results to be somewhat easier to interpret.

If the normalized vectors are parallel and point in the same direction, the dot product of the normalized vectors will be 1.0. If the normalized vectors are parallel and point in opposite directions, the value of the dot product will be -1.0. If the vectors are perpendicular, the dot product of the two vectors will be 0.0 regardless of whether or not they are normalized.

For all orientations, the value of the dot product of normalized vectors will vary between -1.0 and +1.0.

More than three dimensions

You may have it in your mind that the use of mathematical concepts such as the vector dot product are limited to the three dimensions of width, height, and depth. If so, that is a false impression. The vector dot product is very useful for systems with more than three dimensions. In fact, engineers and scientists deal with systems every day that have more than three dimensions. While it usually isn't too difficult to handle the math involved in such systems, we have a very hard time drawing pictures of systems with more than three or four dimensions.

Digital convolution is a good example

I have spent much of my career in digital signal processing where I have routinely dealt with systems having thirty or forty dimensions. For example, in the module titled [Convolution and Frequency Filtering in Java](#) and some other related modules as well, I describe the process of digital *convolution filtering* .

One way to think of digital convolution is that it is a running dot product computation between one vector (*the convolution filter*) and a series of other vectors, each comprised of successive chunks of samples from the incoming data. In non-adaptive systems, the vector that represents the convolution filter usually has a set of fixed coordinate values, and there may be dozens and even hundreds of such coordinates. (*For an adaptive system, the values that define the vector typically change as a function of time or some other parameter.*)

Often, the input data will consist of samples taken from a channel consisting of signal plus noise. The objective is often to design a vector (*the convolution filter*) that is parallel to all of the signal components in the incoming data and is perpendicular to all of the noise components in the incoming data. If that objective is achieved, the noise will be suppressed while the signal will be passed through to the output.

Documentation for the GM02 library

Click [here](#) to download a zip file containing standard javadoc documentation for the library named **GM02** . Extract the contents of the zip file into an empty folder and open the file named **index.html** in your browser to view the documentation.

Although the documentation doesn't provide much in the way of explanatory text (see [Listing 8](#) and the explanations given above) , the documentation does provide a good overview of the organization and structure of the library. You may find it helpful in that regard.

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter 10, Angle between 3D Vectors* .

The homework assignment for the next module is to continue studying that same material.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the programs

I encourage you to copy the code from [Listing 8](#) through [Listing 12](#). Compile the code and execute it in conjunction with the game-math library named **GM02** provided in [Listing 8](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

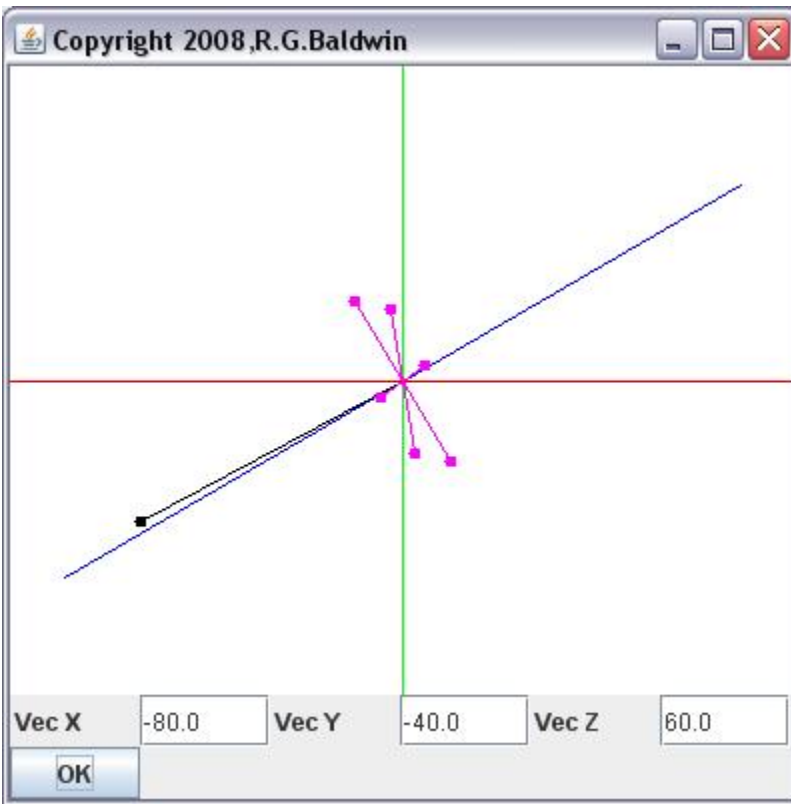
In this module, you learned the fundamentals of the vector dot product in both 2D and 3D. You learned how to update the game-math library to support various aspects of the vector dot product, and you learned how to write 2D and 3D programs that use the vector dot product methods in the game-math library.

What's next?

In the next module, which will be the second part of this two-part miniseries on the vector dot product, you will learn how to use the dot product to compute nine different angles of interest that a vector makes with various elements in 3D space.

You will learn how to use the dot product to find six of the infinite set of vectors that are perpendicular to a given vector as shown in [Figure 16](#).

Figure 16 Six (magenta) vectors that are perpendicular to a given (black) vector.



You will also learn how to use the dot product to perform back-face culling as shown in [Figure 1](#) and [Figure 2](#).

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0145: Getting Started with the Vector Dot Product
- File: Game0145.htm
- Published: 10/21/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listings

Complete listings of the programs discussed in this module are shown in [Listing 8](#) through [Listing 12](#) below

Listing 8 . Source code for the game-math library named GM02.

```
/*GM02.java  
Copyright 2008, R.G.Baldwin  
Revised 02/08/08
```

This is an update to the game-math library named GM01.

The main purpose of this update was to add vector dot product and related capabilities to the library.

Please see the comments at the beginning of the library class named GM01 for a general description of the library.

The following methods are new instance methods of the indicated static top-level classes belonging to the class named GM02.

GM02.ColMatrix2D.dot - compute dot product of two ColMatrix2D objects.

GM02.Vector2D.dot - compute dot product of two Vector2D objects.

GM02.Vector2D.angle - compute angle between two Vector2D objects.

GM02.ColMatrix3D.dot - compute dot product of two ColMatrix3D objects

GM02.Vector3D.dot - compute dot product of two Vector3D objects.

GM02.Vector3D.angle - compute angle between two Vector3D objects.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/  
import java.awt.geom.*;  
import java.awt.*;
```

```

public class GM02{
    //-----
    -----//

    //This method converts a ColMatrix3D obj to a
    // ColMatrix2D object. The purpose is to accept
    // x, y, and z coordinate values and transform
those
    // values into a pair of coordinate values
suitable for
    // display in two dimensions.
    //See
http://local.wasp.uwa.edu.au/~pbourke/geometry/
    // classification/ for technical background on
the
    // transform from 3D to 2D.
    //The transform equations are:
    //  $x_{2d} = x_{3d} + z_{3d} * \cos(\theta)/\tan(\alpha)$ 
    //  $y_{2d} = y_{3d} + z_{3d} * \sin(\theta)/\tan(\alpha)$ ;
    //Let  $\theta = 30$  degrees and  $\alpha = 45$  degrees
    //Then: $\cos(\theta) = 0.866$ 
    //       $\sin(\theta) = 0.5$ 
    //       $\tan(\alpha) = 1$ ;
    //Note that the signs in the above equations
depend
    // on the assumed directions of the angles as
well as
    // the assumed positive directions of the axes.
The
    // signs used in this method assume the
following:
    //      Positive x is to the right.
    //      Positive y is up the screen.
    //      Positive z is protruding out the front of
the
    //      screen.

```

```

    //    The viewing position is above the x axis
and to the
    //    right of the z-y plane.
    public static GM02.ColMatrix2D convert3Dto2D(
GM02.ColMatrix3D data){
        return new GM02.ColMatrix2D(
            data.getData(0) -
0.866*data.getData(2),
            data.getData(1) -
0.50*data.getData(2));
    }//end convert3Dto2D
    //-----
    -----//

    //This method wraps around the translate method
of the
    // Graphics2D class. The purpose is to cause the
    // positive direction for the y-axis to be up
the screen
    // instead of down the screen. When you use this
method,
    // you should program as though the positive
direction
    // for the y-axis is up.
    public static void translate(Graphics2D g2D,
                                double xOffset,
                                double yOffset){
        //Flip the sign on the y-coordinate to change
the
        // direction of the positive y-axis to go up
the
        // screen.
        g2D.translate(xOffset, -yOffset);
    }//end translate
    //-----
    -----//

```

```

    //This method wraps around the drawLine method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void drawLine(Graphics2D g2D,
                                double x1,
                                double y1,
                                double x2,
                                double y2){
        //Flip the sign on the y-coordinate value.
        g2D.drawLine((int)x1, -(int)y1, (int)x2, -
(int)y2);
    }//end drawLine
    //-----
-----//

```

```

    //This method wraps around the fillOval method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void fillOval(Graphics2D g2D,
                                double x,

```

```

                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.fillOval((int)x,-(int)y,(int)width,
(int)height);
    }//end fillOval
    //-----
-----//

```

```

    //This method wraps around the drawOval method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void drawOval(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.drawOval((int)x,-(int)y,(int)width,
(int)height);
    }//end drawOval
    //-----
-----//

```

```

    //This method wraps around the fillRect method
of the
    // Graphics class. The purpose is to cause the
positive

```

```

    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void fillRect(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.fillRect((int)x, -(int)y, (int)width,
(int)height);
    }//end fillRect
    //-----
-----//

```

```

    //An object of this class represents a 2D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix2D{
        double[] data = new double[2];

        public ColMatrix2D(double data0, double data1){
            data[0] = data0;
            data[1] = data1;
        }//end constructor
        //-----
-----//

```

```

//Overridden toString method.
public String toString(){
    return data[0] + "," + data[1];
} //end overridden toString method
//-----
-----//

```

```

public double getData(int index){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        return data[index];
    } //end else
} //end getData method
//-----
-----//

```

```

public void setData(int index, double data){
    if((index < 0) || (index > 1)){
        throw new IndexOutOfBoundsException();
    }else{
        this.data[index] = data;
    } //end else
} //end setData method
//-----
-----//

```

```

//This method overrides the equals method
inherited
// from the class named Object. It compares
the values
// stored in two matrices and returns true if
the
// values are equal or almost equal and
returns false
// otherwise.
public boolean equals(Object obj){

```



```

        if(obj instanceof GM02.ColMatrix2D &&
Math.abs(((GM02.ColMatrix2D)obj).getData(0) -
                                                getData(0)) <=
0.00001 &&
Math.abs(((GM02.ColMatrix2D)obj).getData(1) -
                                                getData(1)) <=
0.00001){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
    -----//

    //Adds one ColMatrix2D object to another
    ColMatrix2D
    // object, returning a ColMatrix2D object.
    public GM02.ColMatrix2D add(GM02.ColMatrix2D
matrix){
        return new GM02.ColMatrix2D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1));
    }//end add
    //-----
    -----//

    //Subtracts one ColMatrix2D object from
    another
    // ColMatrix2D object, returning a ColMatrix2D
    object.
    // The object that is received as an incoming

```

```

        // parameter is subtracted from the object on
which
        // the method is called.
        public GM02.ColMatrix2D subtract(
                                GM02.ColMatrix2D
matrix){
        return new GM02.ColMatrix2D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1));
        }//end subtract
        //-----
        -----//

        //Computes the dot product of two ColMatrix2D
        // objects and returns the result as type
double.
        public double dot(GM02.ColMatrix2D matrix){
        return getData(0) * matrix.getData(0)
                + getData(1) * matrix.getData(1);
        }//end dot
        //-----
        -----//
        }//end class ColMatrix2D

//=====
====//

```

```

        //An object of this class represents a 3D column
matrix.
        // An object of this class is the fundamental
building
        // block for several of the other classes in the
        // library.
        public static class ColMatrix3D{

```

```

double[] data = new double[3];

public ColMatrix3D(
    double data0,double data1,double
data2){
    data[0] = data0;
    data[1] = data1;
    data[2] = data2;
} //end constructor
//-----
-----//

    public String toString(){
        return data[0] + "," + data[1] + "," +
data[2];
    } //end overridden toString method
    //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return data[index];
        } //end else
    } //end getData method
    //-----
    -----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            this.data[index] = data;
        } //end else
    } //end setData method
    //-----

```

```

-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in two matrices and returns true if
the
    // values are equal or almost equal and
returns false
    // otherwise.
    public boolean equals(Object obj){
        if(obj instanceof GM02.ColMatrix3D &&
Math.abs(((GM02.ColMatrix3D)obj).getData(0) -
                                                getData(0)) <=
0.000001 &&
Math.abs(((GM02.ColMatrix3D)obj).getData(1) -
                                                getData(1)) <=
0.000001 &&
Math.abs(((GM02.ColMatrix3D)obj).getData(2) -
                                                getData(2)) <=
0.000001){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
-----//

    //Adds one ColMatrix3D object to another
ColMatrix3D
    // object, returning a ColMatrix3D object.

```

```

        public GM02.ColMatrix3D add(GM02.ColMatrix3D
matrix){
            return new GM02.ColMatrix3D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1),

getData(2)+matrix.getData(2));
        }//end add
        //-----
        -----//

        //Subtracts one ColMatrix3D object from
another
        // ColMatrix3D object, returning a ColMatrix3D
object.
        // The object that is received as an incoming
        // parameter is subtracted from the object on
which
        // the method is called.
        public GM02.ColMatrix3D subtract(
                                GM02.ColMatrix3D
matrix){
            return new GM02.ColMatrix3D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1),
                                getData(2)-
matrix.getData(2));
        }//end subtract
        //-----
        -----//

        //Computes the dot product of two ColMatrix3D
// objects and returns the result as type

```

```

double.
    public double dot(GM02.ColMatrix3D matrix){
        return getData(0) * matrix.getData(0)
            + getData(1) * matrix.getData(1)
            + getData(2) * matrix.getData(2);
    }//end dot
    //-----
-----//
    }//end class ColMatrix3D

//=====
====//

//=====
====//

    public static class Point2D{
        GM02.ColMatrix2D point;

        public Point2D(GM02.ColMatrix2D point)
    {//constructor
        //Create and save a clone of the ColMatrix2D
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix2D
object
        // through use of its set method.
        this.point = new ColMatrix2D(
point.getData(0),point.getData(1));
    }//end constructor
    //-----
-----//

```

```

        public String toString(){
            return point.getData(0) + "," +
point.getData(1);
        }//end toString
        //-----
-----//

        public double getData(int index){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                return point.getData(index);
            }//end else
        }//end getData
        //-----
-----//

        public void setData(int index,double data){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                point.setData(index,data);
            }//end else
        }//end setData
        //-----
-----//

        //This method draws a small circle around the
location
        // of the point on the specified graphics
context.
        public void draw(Graphics2D g2D){
            drawOval(g2D,getData(0)-3,
                        getData(1)+3,6,6);
        }//end draw

```

```

//-----
-----//

//Returns a reference to the ColMatrix2D
object that
// defines this Point2D object.
public GM02.ColMatrix2D getColMatrix(){
    return point;
} //end getColMatrix
//-----
-----//

//This method overrides the equals method
inherited
// from the class named Object. It compares
the values
// stored in the ColMatrix2D objects that
define two
// Point2D objects and returns true if they
are equal
// and false otherwise.
public boolean equals(Object obj){
    if(point.equals(((GM02.Point2D)obj).
getColMatrix())){
        return true;
    }else{
        return false;
    } //end else

} //end overridden equals method
//-----
-----//

//Gets a displacement vector from one Point2D
object
// to a second Point2D object. The vector

```



```

points from
    // the object on which the method is called to
the
    // object passed as a parameter to the method.
Kjell
    // describes this as the distance you would
have to
    // walk along the x and then the y axes to get
from
    // the first point to the second point.
    public GM02.Vector2D getDisplacementVector(
                                                GM02.Point2D
point){
    return new GM02.Vector2D(new
GM02.ColMatrix2D(
                                point.getData(0)-
getData(0),
                                point.getData(1)-
getData(1)));
    }//end getDisplacementVector
    //-----
    -----//

    //Adds a Vector2D to a Point2D producing a
    // new Point2D.
    public GM02.Point2D addVectorToPoint(

GM02.Vector2D vec){
    return new GM02.Point2D(new
GM02.ColMatrix2D(
                                getData(0) +
vec.getData(0),
                                getData(1) +
vec.getData(1)));
    }//end addVectorToPoint
    //-----
    -----//

```

```

of      //Returns a new Point2D object that is a clone
        // the object on which the method is called.
        public Point2D clone(){
            return new Point2D(
                                new
ColMatrix2D(getData(0),getData(1)));
        }//end clone
        //-----
        -----//

```

```

        //The purpose of this method is to rotate a
point
        // around a specified anchor point in the x-y
plane.
        //The rotation angle is passed in as a double
value
        // in degrees with the positive angle of
rotation
        // being counter-clockwise.
        //This method does not modify the contents of
the
        // Point2D object on which the method is
called.
        // Rather, it uses the contents of that object
to
        // instantiate, rotate, and return a new
Point2D
        // object.
        //For simplicity, this method translates the
// anchorPoint to the origin, rotates around
the
        // origin, and then translates back to the
// anchorPoint.
        /*
        See

```

<http://www.ia.hiof.no/~borres/cgraph/math/threed/p-threed.html> for a definition of the equations required to do the rotation.

```
x2 = x1*cos - y1*sin
y2 = x1*sin + y1*cos
*/
public GM02.Point2D rotate(GM02.Point2D
anchorPoint,
                        double angle){
    GM02.Point2D newPoint = this.clone();

    double tempX ;
    double tempY;

    //Translate anchorPoint to the origin
    GM02.Vector2D tempVec =
        new
GM02.Vector2D(anchorPoint.getColMatrix());
    newPoint =

newPoint.addVectorToPoint(tempVec.negate());

    //Rotate around the origin.
    tempX = newPoint.getData(0);
    tempY = newPoint.getData(1);
    newPoint.setData(//new x coordinate
        0,

tempX*Math.cos(angle*Math.PI/180) -
tempY*Math.sin(angle*Math.PI/180));

    newPoint.setData(//new y coordinate
        1,
```

```

tempX*Math.sin(angle*Math.PI/180) +
tempY*Math.cos(angle*Math.PI/180));

    //Translate back to anchorPoint
    newPoint =
newPoint.addVectorToPoint(tempVec);

    return newPoint;

} //end rotate
//-----
-----//

    //Multiplies this point by a scaling matrix
received
    // as an incoming parameter and returns the
scaled
    // point.
    public GM02.Point2D scale(GM02.ColMatrix2D
scale){
        return new GM02.Point2D(new ColMatrix2D(
            getData(0) *
scale.getData(0),
            getData(1) *
scale.getData(1)));
    } //end scale
    //-----
    -----//
} //end class Point2D

//=====
=====//

    public static class Point3D{
        GM02.ColMatrix3D point;

```

```

        public Point3D(GM02.ColMatrix3D point)
{
    //constructor
        //Create and save a clone of the ColMatrix3D
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix3D
object
        // through use of its set method.
        this.point =
            new ColMatrix3D(point.getData(0),
                            point.getData(1),
                            point.getData(2));
    }
}
//-----
-----//

        public String toString(){
            return point.getData(0) + "," +
point.getData(1)
                                + "," +
point.getData(2);
        }
//-----
-----//

        public double getData(int index){
            if((index < 0) || (index > 2)){
                throw new IndexOutOfBoundsException();
            }
            else{
                return point.getData(index);
            }
        }
//-----
//-----

```

```

-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            point.setData(index,data);
        }//end else
    }//end setData
    //-----
-----//

    //This method draws a small circle around the
location
    // of the point on the specified graphics
context.
    public void draw(Graphics2D g2D){

        //Get 2D projection coordinate values.
        ColMatrix2D temp = convert3Dto2D(point);
        drawOval(g2D,temp.getData(0)-3,
                    temp.getData(1)+3,
                    6,
                    6);

    }//end draw
    //-----
-----//

    //Returns a reference to the ColMatrix3D
object that
    // defines this Point3D object.
    public GM02.ColMatrix3D getColMatrix(){
        return point;
    }//end getColMatrix
    //-----
-----//

```

```

        //This method overrides the equals method
inherited
        // from the class named Object. It compares
the values
        // stored in the ColMatrix3D objects that
define two
        // Point3D objects and returns true if they
are equal
        // and false otherwise.
        public boolean equals(Object obj){
            if(point.equals(((GM02.Point3D)obj).

getColMatrix())){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
-----//

        //Gets a displacement vector from one Point3D
object
        // to a second Point3D object. The vector
points from
        // the object on which the method is called to
the
        // object passed as a parameter to the method.
Kjell
        // describes this as the distance you would
have to
        // walk along the x and then the y axes to get
from
        // the first point to the second point.
        public GM02.Vector3D getDisplacementVector(
                                GM02.Point3D

```

```

point){
    return new GM02.Vector3D(new
GM02.ColMatrix3D(
                                point.getData(0)-
getData(0),
                                point.getData(1)-
getData(1),
                                point.getData(2)-
getData(2)));
    }//end getDisplacementVector
    //-----
    -----//

    //Adds a Vector3D to a Point3D producing a
    // new Point3D.
    public GM02.Point3D addVectorToPoint(
GM02.Vector3D vec){
    return new GM02.Point3D(new
GM02.ColMatrix3D(
                                getData(0) +
vec.getData(0),
                                getData(1) +
vec.getData(1),
                                getData(2) +
vec.getData(2)));
    }//end addVectorToPoint
    //-----
    -----//

    //Returns a new Point3D object that is a clone
of
    // the object on which the method is called.
    public Point3D clone(){
    return new Point3D(new
ColMatrix3D(getData(0),

```



```

getData(1),

getData(2)));
    }//end clone
    //-----
-----//

    //The purpose of this method is to rotate a
point
    // around a specified anchor point in the
following
    // order:
    // Rotate around z - rotation in x-y plane.
    // Rotate around x - rotation in y-z plane.
    // Rotate around y - rotation in x-z plane.
    //The rotation angles are passed in as double
values
    // in degrees (based on the right-hand rule)
in the
    // order given above, packaged in an object of
the
    // class GM02.ColMatrix3D. (Note that in this
case,
    // the ColMatrix3D object is simply a
convenient
    // container and it has no significance from a
matrix
    // viewpoint.)
    //The right-hand rule states that if you point
the
    // thumb of your right hand in the positive
direction
    // of an axis, the direction of positive
rotation
    // around that axis is given by the direction
that
    // your fingers will be pointing.

```

```

    //This method does not modify the contents of
the
    // Point3D object on which the method is
called.
    // Rather, it uses the contents of that object
to
    // instantiate, rotate, and return a new
Point3D
    // object.
    //For simplicity, this method translates the
    // anchorPoint to the origin, rotates around
the
    // origin, and then translates back to the
    // anchorPoint.
    /*
    See
http://www.ia.hiof.no/~borres/cgraph/math/threed/
    p-threed.html for a definition of the
equations
    required to do the rotation.
    z-axis

$$x_2 = x_1 \cos \theta - y_1 \sin \theta$$


$$y_2 = x_1 \sin \theta + y_1 \cos \theta$$


    x-axis

$$y_2 = y_1 \cos(\theta) - z_1 \sin(\theta)$$


$$z_2 = y_1 \sin(\theta) + z_1 \cos(\theta)$$


    y-axis

$$x_2 = x_1 \cos(\theta) + z_1 \sin(\theta)$$


$$z_2 = -x_1 \sin(\theta) + z_1 \cos(\theta)$$

    */
    public GM02.Point3D rotate(GM02.Point3D
anchorPoint,
                                GM02.ColMatrix3D
angles){
        GM02.Point3D newPoint = this.clone();

```

```

double tempX ;
double tempY;
double tempZ;

//Translate anchorPoint to the origin
GM02.Vector3D tempVec =
    new
GM02.Vector3D(anchorPoint.getColMatrix());
newPoint =

newPoint.addVectorToPoint(tempVec.negate());

double zAngle = angles.getData(0);
double xAngle = angles.getData(1);
double yAngle = angles.getData(2);

//Rotate around z-axis
tempX = newPoint.getData(0);
tempY = newPoint.getData(1);
newPoint.setData(//new x coordinate
0,

tempX*Math.cos(zAngle*Math.PI/180) -
tempY*Math.sin(zAngle*Math.PI/180));

newPoint.setData(//new y coordinate
1,

tempX*Math.sin(zAngle*Math.PI/180) +
tempY*Math.cos(zAngle*Math.PI/180));

//Rotate around x-axis
tempY = newPoint.getData(1);
tempZ = newPoint.getData(2);

```

```

        newPoint.setData(new y coordinate
                        1,
tempY*Math.cos(xAngle*Math.PI/180) -
tempZ*Math.sin(xAngle*Math.PI/180));

        newPoint.setData(new z coordinate
                        2,
tempY*Math.sin(xAngle*Math.PI/180) +
tempZ*Math.cos(xAngle*Math.PI/180));

        //Rotate around y-axis
        tempX = newPoint.getData(0);
        tempZ = newPoint.getData(2);
        newPoint.setData(new x coordinate
                        0,
tempX*Math.cos(yAngle*Math.PI/180) +
tempZ*Math.sin(yAngle*Math.PI/180));

        newPoint.setData(new z coordinate
                        2,
-
tempX*Math.sin(yAngle*Math.PI/180) +
tempZ*Math.cos(yAngle*Math.PI/180));

        //Translate back to anchorPoint
        newPoint =
newPoint.addVectorToPoint(tempVec);

        return newPoint;

```

```

        }//end rotate
        //-----
-----//

        //Multiplies this point by a scaling matrix
        received
        // as an incoming parameter and returns the
        scaled
        // point.
        public GM02.Point3D scale(GM02.ColMatrix3D
scale){
        return new GM02.Point3D(new ColMatrix3D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1),
                                getData(2) *
scale.getData(2)));
        }//end scale
        //-----
-----//
    }//end class Point3D

//=====
====//

//=====
====//

    public static class Vector2D{
        GM02.ColMatrix2D vector;

        public Vector2D(GM02.ColMatrix2D vector)
        {//constructor
            //Create and save a clone of the ColMatrix2D
            object

```

```

        // used to define the vector to prevent the
vector
        // from being corrupted by a later change in
the
        // values stored in the original ColVector2D
object.
        this.vector = new ColMatrix2D(
vector.getData(0),vector.getData(1));
    }//end constructor
    //-----
    -----//

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1);
    }//end toString
    //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
    //-----
    -----//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData

```

```

//-----
-----//

//This method draws a vector on the specified
graphics
// context, with the tail of the vector
located at a
// specified point, and with a small filled
circle at
// the head.
public void draw(Graphics2D g2D, GM02.Point2D
tail){

    drawLine(g2D,
              tail.getData(0),
              tail.getData(1),
              tail.getData(0)+vector.getData(0),
              tail.getData(1)+vector.getData(1));

    fillOval(g2D,

tail.getData(0)+vector.getData(0)-3,

tail.getData(1)+vector.getData(1)+3,
              6,
              6);
} //end draw
//-----
-----//

//Returns a reference to the ColMatrix2D
object that
// defines this Vector2D object.
public GM02.ColMatrix2D getColMatrix(){
    return vector;
} //end getColMatrix
//-----

```

```

-----//

    //This method overrides the equals method
inherited
    // from the class named Object. It compares
the values
    // stored in the ColMatrix2D objects that
define two
    // Vector2D objects and returns true if they
are equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(vector.equals((
(GM02.Vector2D)obj).getColMatrix())){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
-----//

    //Adds this vector to a vector received as an
incoming
    // parameter and returns the sum as a vector.
    public GM02.Vector2D add(GM02.Vector2D vec){
        return new GM02.Vector2D(new ColMatrix2D(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1)));
    }//end add
    //-----
-----//

```



```

//Returns the length of a Vector2D object.
public double getLength(){
    return Math.sqrt(
        getData(0)*getData(0) +
        getData(1)*getData(1));
} //end getLength
//-----
-----//

//Multiplies this vector by a scale factor
received as
// an incoming parameter and returns the
scaled
// vector.
public GM02.Vector2D scale(Double factor){
    return new GM02.Vector2D(new ColMatrix2D(
        getData(0) *
factor,
        getData(1) *
factor));
} //end scale
//-----
-----//

//Changes the sign on each of the vector
components
// and returns the negated vector.
public GM02.Vector2D negate(){
    return new GM02.Vector2D(new ColMatrix2D(
        -
        -
        getData(0),
        getData(1)));
} //end negate
//-----
-----//

```

```

        //Returns a new vector that points in the same
        // direction but has a length of one unit.
        public GM02.Vector2D normalize(){
            double length = getLength();
            return new GM02.Vector2D(new ColMatrix2D(

getData(0)/length,

getData(1)/length));
        }//end normalize
        //-----
        -----//

        //Computes the dot product of two Vector2D
        // objects and returns the result as type
double.
        public double dot(GM02.Vector2D vec){
            GM02.ColMatrix2D matrixA = getColMatrix();
            GM02.ColMatrix2D matrixB =
vec.getColMatrix();
            return matrixA.dot(matrixB);
        }//end dot
        //-----
        -----//

        //Computes and returns the angle between two
Vector2D
        // objects. The angle is returned in degrees
as type
        // double.
        public double angle(GM02.Vector2D vec){
            GM02.Vector2D normA = normalize();
            GM02.Vector2D normB = vec.normalize();
            double normDotProd = normA.dot(normB);
            return
Math.toDegrees(Math.acos(normDotProd));
        }//end angle

```

```

        //-----
-----//
    }//end class Vector2D

//=====
====//

    public static class Vector3D{
        GM02.ColMatrix3D vector;

        public Vector3D(GM02.ColMatrix3D vector)
    {//constructor
        //Create and save a clone of the ColMatrix3D
        object
        // used to define the vector to prevent the
        vector
        // from being corrupted by a later change in
        the
        // values stored in the original ColMatrix3D
        object.
        this.vector = new
        ColMatrix3D(vector.getData(0),
        vector.getData(1),
        vector.getData(2));
        }//end constructor
        //-----
-----//

        public String toString(){
            return vector.getData(0) + "," +
            vector.getData(1)
            + "," +
            vector.getData(2);
        }//end toString
    }

```

```

//-----
-----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
//-----
-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData
//-----
-----//

    //This method draws a vector on the specified
graphics
    // context, with the tail of the vector
located at a
    // specified point, and with a small circle at
the
    // head.
    public void draw(Graphics2D g2D,GM02.Point3D
tail){

        //Get a 2D projection of the tail
        GM02.ColMatrix2D tail2D =
convert3Dto2D(tail.point);

```

```

        //Get the 3D location of the head
        GM02.ColMatrix3D head =

tail.point.add(this.getColMatrix());

        //Get a 2D projection of the head
        GM02.ColMatrix2D head2D =
convert3Dto2D(head);
        drawLine(g2D,tail2D.getData(0),
                    tail2D.getData(1),
                    head2D.getData(0),
                    head2D.getData(1));

        //Draw a small filled circle to identify the
head.
        fillOval(g2D,head2D.getData(0)-3,
                    head2D.getData(1)+3,
                    6,
                    6);

        }//end draw
        //-----
-----//

        //Returns a reference to the ColMatrix3D
object that
        // defines this Vector3D object.
        public GM02.ColMatrix3D getColMatrix(){
            return vector;
        }//end getColMatrix
        //-----
-----//

        //This method overrides the equals method
inherited
        // from the class named Object. It compares
the values

```

```

        // stored in the ColMatrix3D objects that
define two
        // Vector3D objects and returns true if they
are equal
        // and false otherwise.
        public boolean equals(Object obj){
            if(vector.equals((
(GM02.Vector3D)obj).getColMatrix())){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
-----//

        //Adds this vector to a vector received as an
incoming
        // parameter and returns the sum as a vector.
        public GM02.Vector3D add(GM02.Vector3D vec){
            return new GM02.Vector3D(new ColMatrix3D(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1),
vec.getData(2)+vector.getData(2)));
        }//end add
        //-----
-----//

        //Returns the length of a Vector3D object.
        public double getLength(){
            return Math.sqrt(getData(0)*getData(0) +
                            getData(1)*getData(1) +

```

```

                                getData(2)*getData(2));
        }//end getLength
        //-----
        -----//

        //Multiplies this vector by a scale factor
        received as
        // an incoming parameter and returns the
        scaled
        // vector.
        public GM02.Vector3D scale(Double factor){
            return new GM02.Vector3D(new ColMatrix3D(
                                getData(0) *
factor,                                getData(1) *
factor,                                getData(2) *
factor));
        }//end scale
        //-----
        -----//

        //Changes the sign on each of the vector
        components
        // and returns the negated vector.
        public GM02.Vector3D negate(){
            return new GM02.Vector3D(new ColMatrix3D(
                                -
getData(0),                                -
                                -
getData(1),                                -
                                -
getData(2)));
        }//end negate
        //-----
        -----//

```

```

        //Returns a new vector that points in the same
        // direction but has a length of one unit.
        public GM02.Vector3D normalize(){
            double length = getLength();
            return new GM02.Vector3D(new ColMatrix3D(

getData(0)/length,

getData(1)/length,

getData(2)/length));
        }//end normalize
        //-----
        -----//

        //Computes the dot product of two Vector3D
        // objects and returns the result as type
        double.
        public double dot(GM02.Vector3D vec){
            GM02.ColMatrix3D matrixA = getColMatrix();
            GM02.ColMatrix3D matrixB =
vec.getColMatrix();
            return matrixA.dot(matrixB);
        }//end dot
        //-----
        -----//

        //Computes and returns the angle between two
        Vector3D
        // objects. The angle is returned in degrees
        as type
        // double.
        public double angle(GM02.Vector3D vec){
            GM02.Vector3D normA = normalize();
            GM02.Vector3D normB = vec.normalize();
            double normDotProd = normA.dot(normB);
            return

```



```

Math.toDegrees(Math.acos(normDotProd));
    }//end angle
    //-----
-----//
    }//end class Vector3D

//=====
====//

//=====
====//

    //A line is defined by two points. One is called
the
    // tail and the other is called the head. Note
that this
    // class has the same name as one of the classes
in
    // the Graphics2D class. Therefore, if the class
from
    // the Graphics2D class is used in some future
upgrade
    // to this program, it will have to be fully
qualified.
    public static class Line2D{
        GM02.Point2D[] line = new GM02.Point2D[2];

        public Line2D(GM02.Point2D tail,GM02.Point2D
head){
            //Create and save clones of the points used
to
            // define the line to prevent the line from
being
            // corrupted by a later change in the
coordinate
            // values of the points.

```

```

        this.line[0] = new Point2D(new
GM02.ColMatrix2D(
tail.getData(0),tail.getData(1)));
        this.line[1] = new Point2D(new
GM02.ColMatrix2D(
head.getData(0),head.getData(1)));
    }//end constructor
    //-----
-----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + ","
            + line[1].getData(1);
    }//end toString
    //-----
-----//

    public GM02.Point2D getTail(){
        return line[0];
    }//end getTail
    //-----
-----//

    public GM02.Point2D getHead(){
        return line[1];
    }//end getHead
    //-----
-----//

    public void setTail(GM02.Point2D newPoint){
to        //Create and save a clone of the new point
        // prevent the line from being corrupted by

```

```

a
    // later change in the coordinate values of
the
    // point.
    this.line[0] = new Point2D(new
GM02.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
    }//end setTail
    //-----
-----//

    public void setHead(GM02.Point2D newPoint){
        //Create and save a clone of the new point
to
        // prevent the line from being corrupted by
a
        // later change in the coordinate values of
the
        // point.
        this.line[1] = new Point2D(new
GM02.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
        }//end setHead
        //-----
-----//

        public void draw(Graphics2D g2D){
            drawLine(g2D, getTail().getData(0),
                    getTail().getData(1),
                    getHead().getData(0),
                    getHead().getData(1));
        }//end draw
        //-----
-----//
    }//end class Line2D

```

```
//=====
====//
```

```
//A line is defined by two points. One is called  
the
```

```
// tail and the other is called the head.
```

```
public static class Line3D{
```

```
    GM02.Point3D[] line = new GM02.Point3D[2];
```

```
    public Line3D(GM02.Point3D tail,GM02.Point3D  
head){
```

```
        //Create and save clones of the points used  
to
```

```
        // define the line to prevent the line from  
being
```

```
        // corrupted by a later change in the  
coordinate
```

```
        // values of the points.
```

```
        this.line[0] = new Point3D(new  
GM02.ColMatrix3D(
```

```
tail.getData(0),
```

```
tail.getData(1),
```

```
tail.getData(2)));
```

```
        this.line[1] = new Point3D(new  
GM02.ColMatrix3D(
```

```
head.getData(0),
```

```
head.getData(1),
```

```
head.getData(2)));
```

```
    }//end constructor
```

```

//-----
-----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
                + line[0].getData(1) + ","
                + line[0].getData(2)
                + "\nHead = "
                + line[1].getData(0) + ","
                + line[1].getData(1) + ","
                + line[1].getData(2);
    }//end toString
//-----
-----//

    public GM02.Point3D getTail(){
        return line[0];
    }//end getTail
//-----
-----//

    public GM02.Point3D getHead(){
        return line[1];
    }//end getHead
//-----
-----//

    public void setTail(GM02.Point3D newPoint){
        //Create and save a clone of the new point
to        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the        // point.
        this.line[0] = new Point3D(new
GM02.ColMatrix3D(

```

```

newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
    }//end setTail
    //-----
-----//

    public void setHead(GM02.Point3D newPoint){
        //Create and save a clone of the new point
to        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the        // point.
        this.line[1] = new Point3D(new
GM02.ColMatrix3D(
newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
        }//end setHead
        //-----
-----//

    public void draw(Graphics2D g2D){

        //Get 2D projection coordinates.
        GM02.ColMatrix2D tail =
convert3Dto2D(getTail().point);
        GM02.ColMatrix2D head =

```

```

convert3Dto2D(getHead().point);

        drawLine(g2D, tail.getData(0),
                  tail.getData(1),
                  head.getData(0),
                  head.getData(1));
    }//end draw
    //-----
-----//
} //end class Line3D

//=====
====//

} //end class GM02

```

Listing 9 . Source code for the program named DotProd2D01.

```

/*DotProd2D01.java
Copyright 2008, R.G.Baldwin
Revised 03/04/08

```

Study Kjell through Chapter 8 - Length,
Orthogonality, and
the Column Matrix Dot product.

The purpose of this program is to confirm proper
operation
of the ColMatrix2D.dot method.

The program creates a GUI that allows the user to
enter
the first and second values for a pair of
ColMatrix2D
objects along with a button labeled OK.

When the user clicks the OK button, the dot product between the two ColMatrix2D objects is computed.

The resulting value is converted to four decimal digits and displayed in a text field on the GUI.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/
```

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.event.*;
```

```
class DotProd2D01{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class DotProd2D01  
//=====
```

```
class GUI extends JFrame implements  
ActionListener{  
    //Specify the horizontal and vertical size of a  
    JFrame  
    // object.  
    int hSize = 400;  
    int vSize = 150;  
  
    JTextField colMatA0 = new JTextField("0.707");  
    JTextField colMatA1 = new JTextField("0.707");  
    JTextField colMatB0 = new JTextField("-0.707");  
    JTextField colMatB1 = new JTextField("-0.707");  
    JTextField dotProduct = new JTextField();  
    JButton button = new JButton("OK");
```



```
//-----  
-----//
```

```
GUI(){//constructor  
  
    //Set JFrame size, title, and close operation.  
    setSize(hSize,vSize);  
    setTitle("Copyright 2008,R.G.Baldwin");  
  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    //Instantiate a JPanel that will house the  
user input  
    // components and set its layout manager.  
    JPanel controlPanel = new JPanel();  
    controlPanel.setLayout(new GridLayout(0,4));  
  
    //Add the user input components and  
appropriate labels  
    // to the control panel.  
    controlPanel.add(new JLabel(" colMatA0 = "));  
    controlPanel.add(colMatA0);  
  
    controlPanel.add(new JLabel(" colMatA1 = "));  
    controlPanel.add(colMatA1);  
  
    controlPanel.add(new JLabel(" colMatB0 = "));  
    controlPanel.add(colMatB0);  
  
    controlPanel.add(new JLabel(" colMatB1 = "));  
    controlPanel.add(colMatB1);  
  
    controlPanel.add(new JLabel(" Dot Prod = "));  
    controlPanel.add(dotProduct);
```

```

        controlPanel.add(new JLabel("")); //spacer
        controlPanel.add(button);

        //Add the control panel to the CENTER position
in the
        // JFrame.
        this.getContentPane().add(
BorderLayout.CENTER,controlPanel);
        setVisible(true);

        //Register this object as an action listener
on the
        // button.
        button.addActionListener(this);

    } //end constructor
    //-----
    -----//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){
        //Create two ColMatrix2D objects.
        GM02.ColMatrix2D matrixA = new
GM02.ColMatrix2D(
            Double.parseDouble(colMatA0.getText()),
            Double.parseDouble(colMatA1.getText()));

        GM02.ColMatrix2D matrixB = new
GM02.ColMatrix2D(
            Double.parseDouble(colMatB0.getText()),
            Double.parseDouble(colMatB1.getText()));

        //Compute the dot product.
        double dotProd = matrixA.dot(matrixB);

```

```

        //Eliminate exponential notation in the
display.

        if(Math.abs(dotProd) < 0.001){
            dotProd = 0.0;
        }//end if

        //Convert to four decimal digits and display.
dotProd =((int)(10000*dotProd))/10000.0;
dotProduct.setText("" + dotProd);

    }//end actionPerformed

//=====
====//

} //end class GUI

```

Listing 10 . Source code for the program named DotProd2D02.

```

/*DotProd2D02.java
Copyright 2008, R.G.Baldwin
Revised 03/07/08

```

This program allows the user to experiment with the dot product and the angle between a pair of GM02.Vector2D objects.

Study Kjell through Chapter 9, The Angle Between Two Vectors.

A GUI is provided that allows the user to enter four double values that define each of two

double values that define each of two
GM02.Vector2D

objects. The GUI also provides an OK button as well as two text fields used for display of computed results.

In addition, the GUI provides a 2D drawing area.

When the user clicks the OK button, the program draws the two vectors, one in black and the other in magenta, on the output screen with the tail of each vector located at the origin in 2D space.

The program also displays the values of the dot product of the two vectors and the angle between the two vectors in degrees.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/  
import java.awt.*;  
import javax.swing.*;  
import java.awt.event.*;  
  
class DotProd2D02{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class DotProd2D02  
//=====
```

```

class GUI extends JFrame implements
ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.

    //User input components.
    JTextField vectorAx = new JTextField("50.0");
    JTextField vectorAy = new JTextField("100.0");
    JTextField vectorBx = new JTextField("-100.0");
    JTextField vectorBy = new JTextField("-50.0");
    JTextField dotProduct = new JTextField();
    JTextField angleDisplay = new JTextField();
    JButton button = new JButton("OK");

    //-----
    -----//

    GUI(){//constructor

        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Instantiate a JPanel that will house the
        user input
        // components and set its layout manager.
        JPanel controlPanel = new JPanel().

```

```

        frame.getContentPane().add(controlPanel);
        controlPanel.setLayout(new GridLayout(0,4));

        //Add the user input components and
appropriate labels
        // to the control panel.
        controlPanel.add(new JLabel(" VectorAx = "));
        controlPanel.add(vectorAx);

        controlPanel.add(new JLabel(" VectorAy = "));
        controlPanel.add(vectorAy);

        controlPanel.add(new JLabel(" VectorBx = "));
        controlPanel.add(vectorBx);

        controlPanel.add(new JLabel(" VectorBy = "));
        controlPanel.add(vectorBy);

        controlPanel.add(new JLabel(" Dot Prod = "));
        controlPanel.add(dotProduct);

        controlPanel.add(new JLabel(" Angle (deg) =
"));
        controlPanel.add(angleDisplay);

        controlPanel.add(button);

        //Add the control panel to the SOUTH position
in the
        // JFrame.
        this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

        //Create a new drawing canvas and add it to
the
        // CENTER of the JFrame above the control

```

```

        // CENTER OF THE OFF-SCREEN IMAGE ABOVE THE CONTROL
panel.

        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER, myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth, osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM02.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

        //Register this object as an action listener
on the
        // button.
button.addActionListener(this);

        //Cause the overridden paint method belonging

```

```

        //erase the overridden paint method belonging
to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    -----//

    //This method is used to draw orthogonal 2D axes
on the
    // off-screen image that intersect at the
origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Erase the screen
        g2D.setColor(Color.WHITE);
        GM02.fillRect(g2D, -osiWidth/2, osiHeight/2,
osiWidth, osiHeight);

        //Draw x-axis in RED
        g2D.setColor(Color.RED);
        GM02.Point2D pointA = new GM02.Point2D(
            new GM02.ColMatrix2D(-
osiWidth/2, 0));
        GM02.Point2D pointB = new GM02.Point2D(
            new
GM02.ColMatrix2D(osiWidth/2, 0));
        new GM02.Line2D(pointA, pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA = new GM02.Point2D(
            new GM02.ColMatrix2D(0, -
osiHeight/2));
        pointB = new GM02.Point2D(
            new

```



```

        new
GM02.ColMatrix2D(0,osiHeight/2));
        new GM02.Line2D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method
    //-----
    -----//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){

        //Erase the off-screen image and draw the
axes.
        setCoordinateFrame(g2D);

        //Create two ColMatrix2D objects based on the
user
        // input values.
        GM02.ColMatrix2D matrixA = new
GM02.ColMatrix2D(

Double.parseDouble(vectorAx.getText()),

Double.parseDouble(vectorAy.getText())));

        GM02.ColMatrix2D matrixB = new
GM02.ColMatrix2D(

Double.parseDouble(vectorBx.getText()),

Double.parseDouble(vectorBy.getText())));

        //Use the ColMatrix2D objects to create two
Vector2D
        // objects

```

```

// Objects.
GM02.Vector2D vecA = new
GM02.Vector2D(matrixA);
GM02.Vector2D vecB = new
GM02.Vector2D(matrixB);

//Draw the two vectors with their tails at the
origin.
g2D.setColor(Color.BLACK);
vecA.draw(
    g2D,new GM02.Point2D(new
GM02.ColMatrix2D(0,0)));

g2D.setColor(Color.MAGENTA);
vecB.draw(
    g2D,new GM02.Point2D(new
GM02.ColMatrix2D(0,0)));

//Compute the dot product of the two vectors.
double dotProd = vecA.dot(vecB);

//Eliminate exponential notation in the
display.
if(Math.abs(dotProd) < 0.001){
    dotProd = 0.0;
}

//Convert to four decimal digits and display.
dotProd =((int)(10000*dotProd))/10000.0;
dotProduct.setText("" + dotProd);

//Compute the angle between the two vectors.
double angle = vecA.angle(vecB);

//Eliminate exponential notation in the
display.
if(Math.abs(angle) < 0.001){

```

```

        if (Math.abs(angle) > 0.001){
            angle = 0.0;
        } //end if

        //Convert to four decimal digits and display.
        angle = ((int)(10000*angle))/10000.0;
        angleDisplay.setText("" + angle);

        myCanvas.repaint(); //Copy off-screen image to
        canvas.
    } //end actionPerformed

//=====
====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
        will be
        // called when the JFrame and the Canvas
        appear on the
        // screen or when the repaint method is called
        on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        } //end overridden paint()

    } //end inner class MyCanvas

} //end class GUI

```

Listing 11 . Source code for the program named DotProd3D01.

```
/*DotProd3D01.java
Copyright 2008, R.G.Baldwin
Revised 03/04/08
```

Study Kjell through Chapter 8 - Length,
Orthogonality, and
the Column Matrix Dot product.

The purpose of this program is to confirm proper
operation
of the ColMatrix3D.dot method.

The program creates a GUI that allows the user to
enter
three values for each of a pair of ColMatrix3D
objects
along with a button labeled OK.

When the user clicks the OK button, the dot
product
between the two ColMatrix3D objects is computed.
The
resulting value is converted to four decimal
digits and
displayed in a text field on the GUI.

Tested using JDK 1.6 under WinXP.

```
*****
*****/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class DotProd3D01{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
} //end controlling class DotProd3D01
```

```

} //end controlling class DOTPROD3D01
//=====
=====//

class GUI extends JFrame implements
ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 470;
    int vSize = 150;

    JTextField colMatA0 = new JTextField("0.57735");
    JTextField colMatA1 = new JTextField("0.57735");
    JTextField colMatA2 = new JTextField("0.57735");

    JTextField colMatB0 = new
    JTextField("-0.57735");
    JTextField colMatB1 = new
    JTextField("-0.57735");
    JTextField colMatB2 = new
    JTextField("-0.57735");

    JTextField dotProduct = new JTextField();
    JButton button = new JButton("OK");

    //-----
    -----//

    GUI(){//constructor

        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```

        //Instantiate a JPanel that will house the
user input
        // components and set its layout manager.
        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new GridLayout(0,6));

        //Add the user input components and
appropriate labels
        // to the control panel.
        controlPanel.add(new JLabel(" colMatA0 = "));
        controlPanel.add(colMatA0);

        controlPanel.add(new JLabel(" colMatA1 = "));
        controlPanel.add(colMatA1);

        controlPanel.add(new JLabel(" colMatA2 = "));
        controlPanel.add(colMatA2);

        controlPanel.add(new JLabel(" colMatB0 = "));
        controlPanel.add(colMatB0);

        controlPanel.add(new JLabel(" colMatB1 = "));
        controlPanel.add(colMatB1);

        controlPanel.add(new JLabel(" colMatB2 = "));
        controlPanel.add(colMatB2);

        controlPanel.add(new JLabel(" Dot Prod = "));
        controlPanel.add(dotProduct);

        controlPanel.add(new JLabel("")); //spacer
        controlPanel.add(button);

        //Add the control panel to the CENTER position
in the
        // JFrame.
        this.getContentPane().add(

```

```

        this.getContentPane().add(
BorderLayout.CENTER,controlPanel);
        setVisible(true);

        //Register this object as an action listener
on the
        // button.
        button.addActionListener(this);

    }//end constructor
    //-----
    -----//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){
        //Create two ColMatrix3D objects.
        GM02.ColMatrix3D matrixA = new
GM02.ColMatrix3D(
            Double.parseDouble(colMatA0.getText()),
            Double.parseDouble(colMatA1.getText()),
            Double.parseDouble(colMatA2.getText()));

        GM02.ColMatrix3D matrixB = new
GM02.ColMatrix3D(
            Double.parseDouble(colMatB0.getText()),
            Double.parseDouble(colMatB1.getText()),
            Double.parseDouble(colMatB2.getText()));

        //Compute the dot product.
        double dotProd = matrixA.dot(matrixB);

        //Eliminate exponential notation in the
display.
        if(Math.abs(dotProd) < 0.001){
            dotProd = 0.0;

```

```

        dotProd = 0.0;
    }//end if

    //Convert to four decimal digits and display.
    dotProd = ((int)(10000*dotProd))/10000.0;
    dotProduct.setText("" + dotProd);

} //end actionPerformed

//=====
====//

} //end class GUI

```

Listing 12 . Source code for the program named DotProd3D02.

```

/*DotProd3D02.java
Copyright 2008, R.G.Baldwin
Revised 03/07/08

```

This program allows the user to experiment with the dot product and the angle between a pair of GM02.Vector3D objects.

Study Kjell through Chapter 10, Angle between 3D Vectors.

A GUI is provided that allows the user to enter six double values that define each of two GM02.Vector3D objects. The GUI also provides an OK button as well as two text fields used for display of computed results.

In addition, the GUI provides a 3D drawing area.

When the user clicks the OK button, the program draws the two vectors, one black and the other in magenta, on the output screen with the tail of each vector located at the origin in 3D space.

The program also displays the values of the dot product of the two vectors and the angle between the two vectors in degrees.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/
```

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.event.*;
```

```
class DotProd3D02{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main  
}//end controlling class DotProd3D02  
//=====
```

```
class GUI extends JFrame implements  
ActionListener{  
    //Specify the horizontal and vertical size of a  
    JFrame  
    // object.  
    int hSize = 400.
```

```

int hSize = 400;
int vSize = 400;
Image osi;//an off-screen image
int osiWidth;//off-screen image width
int osiHeight;//off-screen image height
MyCanvas myCanvas;//a subclass of Canvas
Graphics2D g2D;//off-screen graphics context.

//User input components.
JTextField vecAx = new JTextField("50.0");
JTextField vecAy = new JTextField("100.0");
JTextField vecAz = new JTextField("0.0");
JTextField vecBx = new JTextField("-100.0");
JTextField vecBy = new JTextField("-50.0");
JTextField vecBz = new JTextField("0.0");
JTextField dotProduct = new JTextField();
JTextField angleDisplay = new JTextField();
JButton button = new JButton("OK");

//-----
-----//

GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate a JPanel that will house the
user input
    // components and set its layout manager.
JPanel controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(0,6));

    //Add the user input components and
appropriate labels

```

```

appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" VecAx = "));
    controlPanel.add(vecAx);

    controlPanel.add(new JLabel(" VecAy = "));
    controlPanel.add(vecAy);

    controlPanel.add(new JLabel(" VecAz = "));
    controlPanel.add(vecAz);

    controlPanel.add(new JLabel(" VecBx = "));
    controlPanel.add(vecBx);

    controlPanel.add(new JLabel(" VecrBy = "));
    controlPanel.add(vecBy);

    controlPanel.add(new JLabel(" VecBz = "));
    controlPanel.add(vecBz);

    controlPanel.add(new JLabel(" Dot Prod = "));
    controlPanel.add(dotProduct);

    controlPanel.add(new JLabel(" Ang(deg)"));
    controlPanel.add(angleDisplay);

    controlPanel.add(new JLabel("")); //spacer

    controlPanel.add(button);

    //Add the control panel to the SOUTH position
in the
    // JFrame.
    this.getContentPane().add(
BorderLayout.SOUTH, controlPanel);

```

```

        //Create a new drawing canvas and add it to
the
        // CENTER of the JFrame above the control
panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(

BorderLayout.CENTER,myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM02.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

        //Register this object as an action listener
on the
        // button

```

```

        // success.
        button.addActionListener(this);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    -----//

    //This method is used to draw orthogonal 3D axes
on the
    // off-screen image that intersect at the
origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Erase the screen
        g2D.setColor(Color.WHITE);
        GM02.fillRect(g2D, -osiWidth/2, osiHeight/2,

osiWidth, osiHeight);

        //Draw x-axis in RED
        g2D.setColor(Color.RED);
        GM02.Point3D pointA = new GM02.Point3D(
            new GM02.ColMatrix3D(-
osiWidth/2, 0, 0));
        GM02.Point3D pointB = new GM02.Point3D(
            new
GM02.ColMatrix3D(osiWidth/2, 0, 0));
        new GM02.Line3D(pointA, pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA = new GM02.Point3D(
            new GM02.ColMatrix3D(0, -

```

```

        new GM02.ColMatrix3D(0,
osiHeight/2,0));
        pointB = new GM02.Point3D(
            new
GM02.ColMatrix3D(0,osiHeight/2,0));
        new GM02.Line3D(pointA,pointB).draw(g2D);

        //Draw z-axis in BLUE. Make its length the
same as the
        // length of the x-axis.
        g2D.setColor(Color.BLUE);
        pointA = new GM02.Point3D(
            new GM02.ColMatrix3D(0,0,-
osiWidth/2));
        pointB = new GM02.Point3D(
            new
GM02.ColMatrix3D(0,0,osiWidth/2));
        new GM02.Line3D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method
    //-----
    -----//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){

        //Erase the off-screen image and draw the
axes.
        setCoordinateFrame(g2D);

        //Create two ColMatrix3D objects based on the
user
        // input values.
        GM02.ColMatrix3D matrixA = new
GM02.ColMatrix3D(

```

```
Double.parseDouble(vecAx.getText()),  
Double.parseDouble(vecAy.getText()),  
Double.parseDouble(vecAz.getText())));
```

```
    GM02.ColMatrix3D matrixB = new  
GM02.ColMatrix3D(  
Double.parseDouble(vecBx.getText()),  
Double.parseDouble(vecBy.getText()),  
Double.parseDouble(vecBz.getText())));
```

```
    //Use the ColMatrix3D objects to create two  
Vector3D  
    // objects.  
    GM02.Vector3D vecA = new  
GM02.Vector3D(matrixA);  
    GM02.Vector3D vecB = new  
GM02.Vector3D(matrixB);  
  
    //Draw the two vectors with their tails at the  
origin.  
    g2D.setColor(Color.BLACK);  
    vecA.draw(g2D, new GM02.Point3D(  
new  
GM02.ColMatrix3D(0,0,0)));  
    g2D.setColor(Color.MAGENTA);  
    vecB.draw(g2D, new GM02.Point3D(  
new  
GM02.ColMatrix3D(0,0,0)));
```

```
    //Compute the dot product of the two vectors.  
    double dotProd = vecA.dot(vecB);
```

```

        double dotProd = vecA.dot(vecB);

        //Eliminate exponential notation in the
display.
        if(Math.abs(dotProd) < 0.001){
            dotProd = 0.0;
        }//end if

        //Convert to four decimal digits and display.
dotProd = ((int)(10000*dotProd))/10000.0;
dotProduct.setText("" + dotProd);

        //Compute the angle between the two vectors.
double angle = vecA.angle(vecB);

        //Eliminate exponential notation in the
display.
        if(Math.abs(angle) < 0.001){
            angle = 0.0;
        }//end if

        //Convert to four decimal digits and display.
angle = ((int)(10000*angle))/10000.0;
angleDisplay.setText("" + angle);

        myCanvas.repaint();//Copy off-screen image to
canvas.
    }//end actionPerformed

//=====

====//

        //This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method
will be

```



```

// called when the JFrame and the Canvas
appear on the
// screen or when the repaint method is called
on the
// Canvas object.
//The purpose of this method is to display the
// off-screen image on the screen.
public void paint(Graphics g){
    g.drawImage(osi,0,0,this);
} //end overridden paint()

} //end inner class MyCanvas

} //end class GUI

```

Exercises

Exercise 1

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that behaves as follows.

When the program starts running, an image similar to [Figure 17](#) appears on the screen. Each of the text fields is blank and the drawing area at the top is also blank.

Each time you click the **Replot** button, the program generates three random values in the general range of -128 to 127. The first two values are used as the x and y values for a vector. The two values are displayed in the fields labeled **VectorAx** and **VectorAy** . Also, the two values are used to create and draw a black vector with its tail at the origin as shown in [Figure 17](#).

The third random value is used as the x-value for a second vector. It is displayed in the field labeled **VectorBx** . A y-value is computed that will

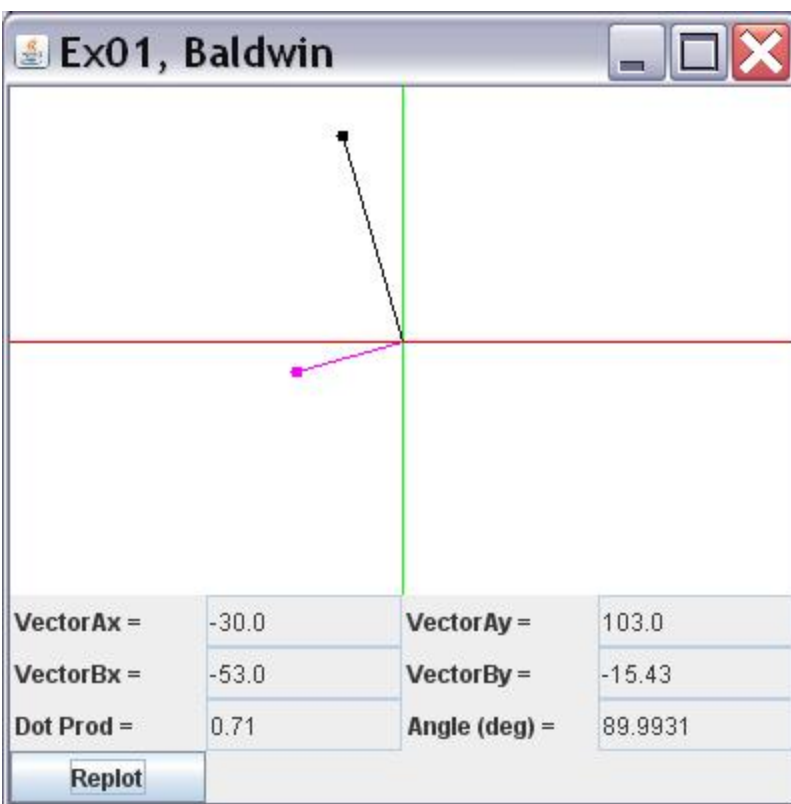
cause that vector to be perpendicular to the black vector. That value is displayed in the field labeled **VectorBy** and the two values are used to draw the magenta vector shown in [Figure 17](#).

The dot product between the two vectors is computed and displayed in the field labeled **Dot Prod** . The angle between the two vectors is computed and displayed in the field labeled **Angle (deg)** .

If the two vectors are perpendicular, the dot product should be close to zero and the angle should be very close to 90 degrees.

Cause your name to appear in the screen output in some manner.

Figure 17 Output from Exercise 1.



Exercise 2

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that behaves as follows.

When the program starts running, an image similar to [Figure 18](#) appears on the screen. Each of the text fields is blank and the drawing area at the top is also blank.

Each time you click the **Plot** button, the program generates three random values in the general range of -128 to 127. The first two values are used as the x and y values for a 3D vector. (Set the z-value for the vector to 0.0.) The three values are displayed in the fields labeled **VecAx** , **VecAy** , and **VecAz** . Also, the three values are used to create and draw a black 3D vector with its tail at the origin as shown in [Figure 18](#).

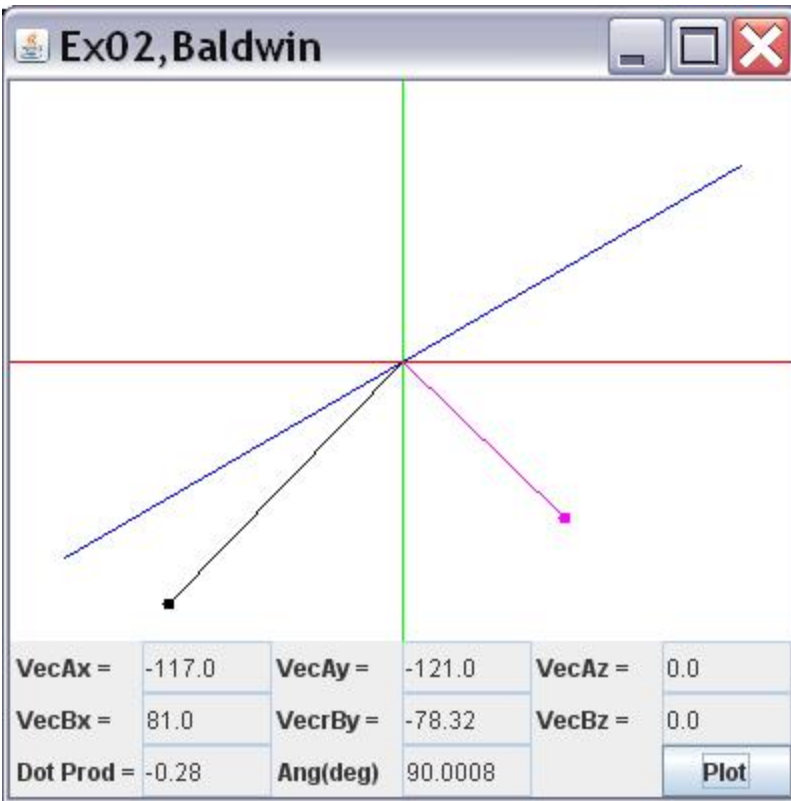
The third random value is used as the x value for a second 3D vector. (Set the z-value for the vector to 0.0.) Those two values are displayed in the fields labeled **VecBx** and **VecBz** . A y-value is computed that will cause that vector to be perpendicular to the black vector. That value is displayed in the field labeled **VecBy** and the three values are used to draw the magenta vector shown in [Figure 18](#).

The dot product between the two vectors is computed and displayed in the field labeled **Dot Prod** . The angle between the two vectors is computed and displayed in the field labeled **Angle (deg)** .

If the two vectors are perpendicular, the dot product should be close to zero and the angle should be very close to 90 degrees.

Cause your name to appear in the screen output in some manner.

Figure 18 Output from Exercise 2.



Exercise 3

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that behaves as follows.

When the program starts running, an image similar to [Figure 19](#) appears on the screen. Each of the text fields is blank and the drawing area at the top is also blank.

Each time you click the **Plot** button, the program generates five random values in the general range of -128 to 127. The first three values are used as the x, y, and z values for a vector. The three values are displayed in the fields labeled **VecAx** , **VecAy** , and **VecAz** . Also, the three values are used to create and draw a black vector with its tail at the origin as shown in [Figure 19](#).

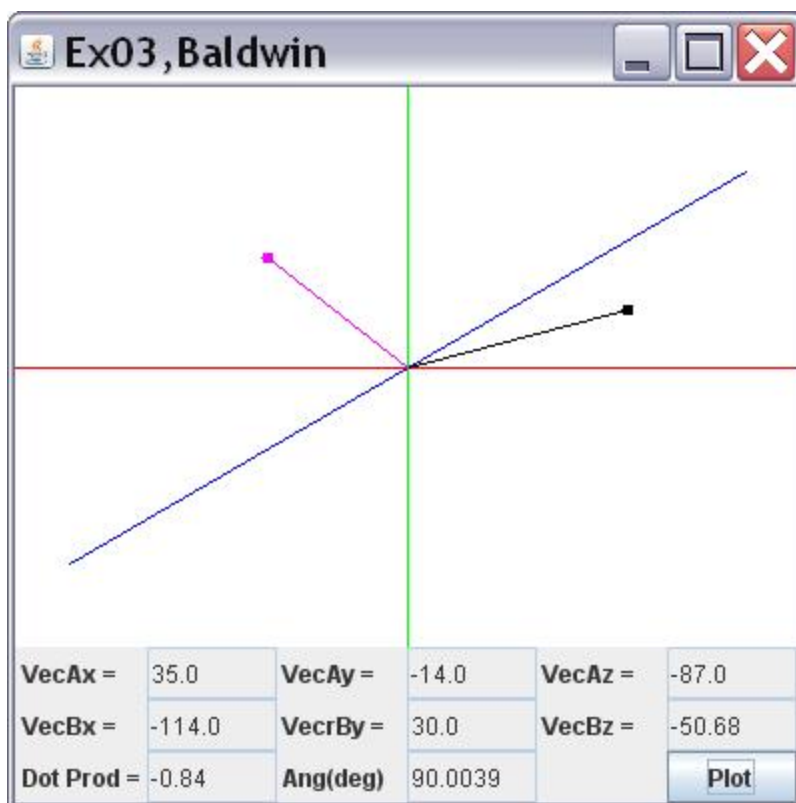
The fourth and fifth random values are used as the x and y values for a second vector. They are displayed in the fields labeled **VecBx** and **VecBy** . A z-value is computed that will cause that vector to be perpendicular to the black vector. That value is displayed in the field labeled **VecBz** and the three values are used to draw the magenta vector shown in [Figure 19](#).

The dot product between the two vectors is computed and displayed in the field labeled **Dot Prod** . The angle between the two vectors is computed and displayed in the field labeled **Angle (deg)** .

If the two vectors are perpendicular, the dot product should be close to zero and the angle should be very close to 90 degrees.

Cause your name to appear in the screen output in some manner.

Figure 19 Output from Exercise 3.



-end-

GAME 2302-0150: Applications of the Vector Dot Product

Learn how to use the dot product to compute nine different angles of interest that a vector makes with various elements in 3D space. Also learn how to use the dot product to find six of the infinite set of vectors that are perpendicular to a given vector, and how to use the dot product to perform back-face culling of an image.

Table of Contents

- [Preface](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Preview](#)
- [Discussion and sample code](#)
 - [The game-math library named GM02](#)
 - [The program named DotProd3D05](#)
 - [The program named DotProd3D06](#)
 - [The program named DotProd3D04](#)
 - [The program named DotProd3D03](#)
- [Homework assignment](#)
- [Run the program](#)
- [Summary](#)
- [What's next?](#)
- [Miscellaneous](#)
- [Complete program listing](#)
- [Exercises](#)
 - [Exercise 1](#)
 - [Exercise 2](#)

Preface

This module is one in a collection of modules designed for teaching *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX.

What you have learned

In the previous module, which was the first part of a two-part miniseries on the vector dot product, you learned the fundamentals of the vector dot product in both 2D and 3D. You learned how to update the game-math library to support various aspects of the vector dot product, and you learned how to write 2D and 3D programs that use the vector dot product methods in the game-math library.

What you will learn

In this module, you will learn how to apply the vector dot product to three different applications. You will learn

- how to use the dot product to compute nine different angles of interest that a vector makes with various elements in 3D space,
- how to use the dot product to find six somewhat unique vectors of the infinite set of vectors that are perpendicular to a given vector as shown in [Figure 4](#), and
- how to use the dot product to perform back-face culling to convert the image shown in [Figure 1](#) to the image shown in [Figure 2](#).

Figure 1 - A 3D image before back-face culling.

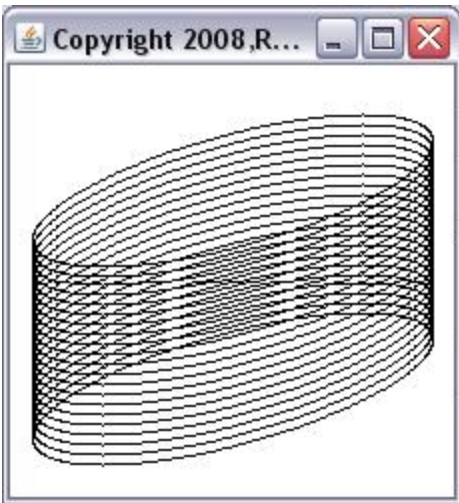
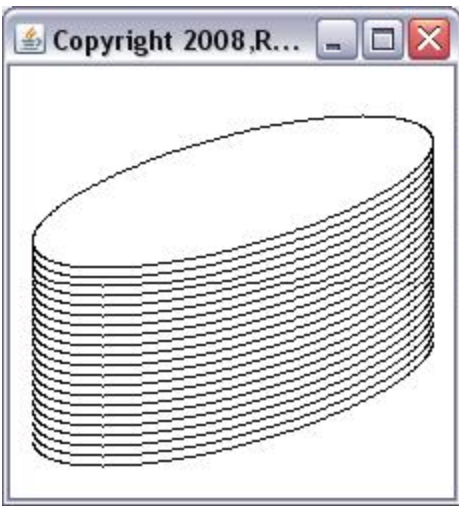


Figure 2 - The 3D image after back-face culling.



Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). A 3D image before back-face culling.

- [Figure 2](#). The 3D image after back-face culling.
- [Figure 3](#). Screen shot of the output from the program named DotProd3D05.
- [Figure 4](#). Six (magenta) vectors that are perpendicular to a given (black) vector.
- [Figure 5](#). Screen output when one coordinate has a value of zero.
- [Figure 6](#). A general formulation of 3D vector perpendicularity.
- [Figure 7](#). Output from Exercise 1.
- [Figure 8](#). Output from Exercise 2.

Listings

- [Listing 1](#). Beginning of the actionPerformed method in the program named DotProd3D05.
- [Listing 2](#). Create ColMatrix3D objects that represent projections.
- [Listing 3](#). Create and draw vectors.
- [Listing 4](#). Compute and display the nine angles.
- [Listing 5](#). Beginning of the actionPerformed method for the program named DotProd3D06.
- [Listing 6](#). Remainder of the actionPerformed method.
- [Listing 7](#). The method named drawTheCylinder in DotProd3D04.
- [Listing 8](#). The method named drawTheCylinder in DotProd3D03.
- [Listing 9](#). Source code for the game-math library named GM02.
- [Listing 10](#). Source code for the program named DotProd3D05.
- [Listing 11](#). Source code for the program named DotProd3D06.
- [Listing 12](#). Source code for the program named DotProb3D04.
- [Listing 13](#). Source code for the program named DotProb3D03.

Preview

This module will build on what you learned about the vector dot product in the earlier module titled [GAME 2302-0145: Getting Started with the Vector Dot Product](#). In that module, you learned some of the theory behind the dot product. In this module, you will learn how to use the dot-product methods of the game-math library to write several applications. I will present and explain the following four programs:

- **DotProd3D05** - Demonstrates how the dot product can be used to compute nine different angles of interest that a vector makes with various elements in 3D space.
- **DotProd3D06** - Demonstrates the use of the dot product to find six somewhat unique vectors of the infinite set of vectors that are perpendicular to a given vector. (See [Figure 4.](#))
- **DotProd3D04** - Draws the same 3D object as the one drawn in DotProd3D03 but without the benefit of back-face culling. (See [Figure 1.](#))
- **DotProd3D03** - Demonstrates use of the vector dot product for back-face culling. (See [Figure 2.](#))

I will also provide exercises for you to complete on your own at the end of the module. The exercises will concentrate on the material that you have learned in this module and previous modules.

Discussion and sample code

The game-math library named GM02

The game-math library has not been modified since the previous module. Therefore, there is nothing new to discuss and explain insofar as the library is concerned. For your convenience, a complete listing of the source code for the library is provided in [Listing 9](#) near the end of the module.

A link to a zip file containing documentation for the library is provided in the earlier module titled [GAME 2302-0145: Getting Started with the Vector Dot Product](#).

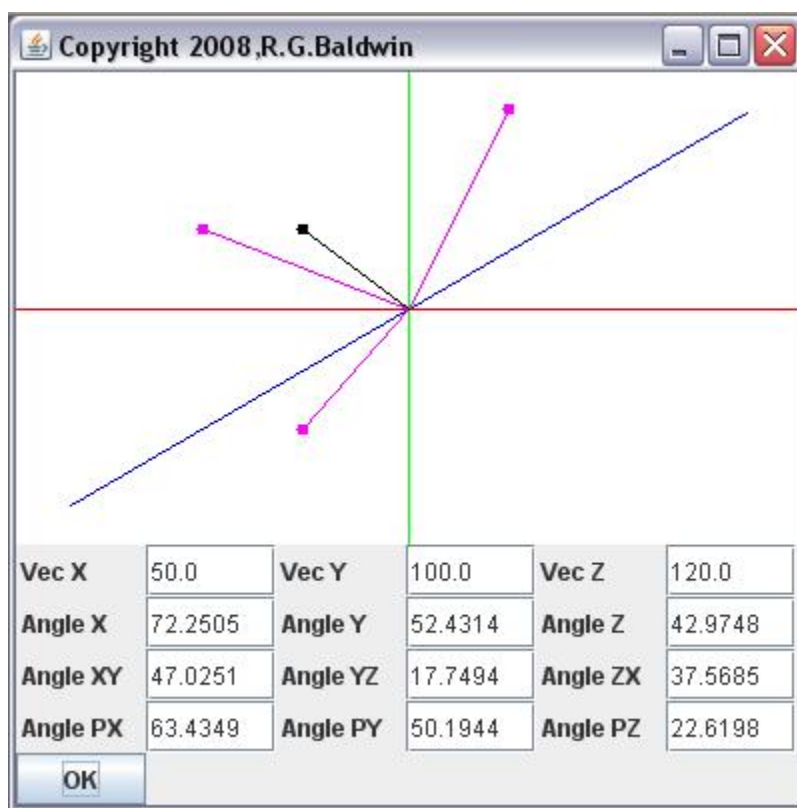
The program named DotProd3D05

In order to understand this and the following programs, you need to understand the material in the Kjell tutorial through *Chapter 10, Angle between 3D Vectors*.

Game programming frequently requires the determination of angles of various types. The purpose of this program is to demonstrate how the dot product can be used to compute nine different angles of interest that a vector makes with various elements in 3Dspace.

[Figure 3](#) shows a screen shot of the graphical user interface provided by this program.

Figure 3 - Screen shot of the output from the program named DotProd3D05.



Angles relative to the axes

First, the program computes and displays the angle between a user-specified vector (*drawn in black in [Figure 3](#)*) and each of the X, Y, and Z axes. These values are displayed with the labels **Angle X** , **Angle Y** , and **Angle Z** in [Figure 3](#).

Angles relative to the XY, YZ, and ZX planes

Then the program computes and displays the angle between the vector and each of the **XY** , **YZ** , and **ZX** planes. In this case, the program computes the smallest possible angle by projecting the vector onto the plane and then computing the angle between the vector and its projection. These values are displayed with the labels **Angle XY** , **Angle YZ** , and **Angle ZX** .

Angles of projections relatives to the axes

Finally, the program computes and displays the angle between the projection of the vector on each of the three planes and one of the axes that defines each plane. The angle between the projection and the other axis that defines the plane is 90 degrees minus the computed angle. Specifically the values that are computed and displayed are:

- Projection onto the **XY** plane relative to the x-axis, displayed with the label **Angle PX** .
- Projection onto the **YZ** plane relative to the y-axis, displayed with the label **Angle PY** .
- Projection onto the **ZX** plane relative to the z-axis, displayed with the label **Angle PZ** .

The graphical user interface

All angles are reported as positive angles in degrees. As you can see in [Figure 3](#), a GUI is provided that allows the user to enter three **double** values that define a **GM02.Vector3D** object. The GUI also provides an **OK** button as well as nine text fields that are used to display the computed results described above. In addition, the GUI provides a 3D drawing area.

When the user clicks the **OK** button, the program draws the user-specified vector in black with the tail located at the origin in 3D space. It also draws the projection of that vector in magenta on each of the **XY** , **YZ** , AND **ZX** planes.

Very similar to previously-explained code

Much of the code in this program is very similar to code that I have explained in previous modules. I won't repeat those explanations here. Most

of the new code is contained in the method named **actionPerformed** , so I will explain the code in that method. A complete listing of this program is provided in [Listing 10](#) near the end of the module.

Beginning of the actionPerformed method in the program named DotProd3D05

[Listing 1](#) shows the beginning of the **actionPerformed** method. This method is called to respond to a click on the **OK** button shown in [Figure 3](#).

Listing 1 . Beginning of the actionPerformed method in the program named DotProd3D05.

Listing 1 . Beginning of the actionPerformed method in the program named DotProd3D05.

```
public void actionPerformed(ActionEvent e){  
    //Erase the off-screen image and draw the  
axes.  
    setCoordinateFrame(g2D);  
  
    //Create one ColMatrix3D object based on  
the user  
    // input values.  
    GM02.ColMatrix3D matrixA = new  
GM02.ColMatrix3D(  
Double.parseDouble(vecX.getText()),  
Double.parseDouble(vecY.getText()),  
Double.parseDouble(vecZ.getText())));  
  
    //Create ColMatrix3D objects that  
represent each of  
    // the three axes.  
    GM02.ColMatrix3D matrixX =  
new  
GM02.ColMatrix3D(1,0,0);  
    GM02.ColMatrix3D matrixY =  
new  
GM02.ColMatrix3D(0,1,0);  
    GM02.ColMatrix3D matrixZ =  
new  
GM02.ColMatrix3D(0,0,1);
```

You have seen code similar to that in [Listing 1](#) many times before. Therefore, this code should not require further explanation beyond the embedded comments.

Create ColMatrix3D objects that represent projections

[Listing 2](#) creates **ColMatrix3D** objects that represent the projection of the user-specified vector onto each of the three planes.

Listing 2 . Create ColMatrix3D objects that represent projections.

Listing 2 . Create ColMatrix3D objects that represent projections.

```
GM02.ColMatrix3D matrixXY = new
GM02.ColMatrix3D(
    Double.parseDouble(vecX.getText()),
    Double.parseDouble(vecY.getText()),
    0);

GM02.ColMatrix3D matrixYZ = new
GM02.ColMatrix3D(
    0,
    Double.parseDouble(vecY.getText()),
    Double.parseDouble(vecZ.getText()));

GM02.ColMatrix3D matrixZX = new
GM02.ColMatrix3D(
    Double.parseDouble(vecX.getText()),
    0,
    Double.parseDouble(vecZ.getText()));
```

Although the actual code in [Listing 2](#) should be familiar to you, the application may be new. The important thing to note is that the projection onto each of the three planes in [Listing 2](#) is accomplished by *setting one of the coordinate values to zero* for each projection. For example, in order to project the vector onto the **XY** plane, the value for the z-axis coordinate is set to zero as shown in [Listing 2](#).

Create and draw vectors

[Listing 3](#) uses the **ColMatrix3D** objects created in [Listing 1](#) and [Listing 2](#) to create and draw vectors based on those **ColMatrix3D** objects.

Listing 3 . Create and draw vectors.

```
//Use the ColMatrix3D objects to create
Vector3D
// objects representing the user-specified
vector and
// each of the axes.
GM02.Vector3D vecA = new
GM02.Vector3D(matrixA);
GM02.Vector3D vecX = new
GM02.Vector3D(matrixX);
GM02.Vector3D vecY = new
GM02.Vector3D(matrixY);
GM02.Vector3D vecZ = new
GM02.Vector3D(matrixZ);

//Create Vector3D objects that represent
the
// projection of the user-specified vector
on each of
// the planes.
GM02.Vector3D vecXY = new
GM02.Vector3D(matrixXY);
GM02.Vector3D vecYZ = new
GM02.Vector3D(matrixYZ);
GM02.Vector3D vecZX = new
GM02.Vector3D(matrixZX);
```

Listing 3 . Create and draw vectors.

```
//Draw the projection of the user
specified vector on
// each of the three planes.
g2D.setColor(Color.MAGENTA);
vecXY.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
vecYZ.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
vecZX.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));

//Draw the user-specified vector with its
tail at the
// origin.
g2D.setColor(Color.BLACK);
vecA.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
```

Once the **ColMatrix3D** objects have been created to serve each specific purpose, there is nothing new in [Listing 3](#). Therefore, no explanation of the code beyond the embedded comments should be necessary.

Compute and display the nine angles

[Listing 4](#) calls the **angle** method of the **GM02.Vector3D** class nine times in succession, using references to the vectors created in [Listing 3](#), to compute and display the nine angle values shown in [Figure 3](#).

Listing 4 . Compute and display the nine angles.

```
the //Compute and display the angle relative to
    // x-axis.
    double angle = vecA.angle(vecX);
    angleX.setText("" +
prepareForDisplay(angle));
```

```
the //Compute and display the angle relative to
    // y-axis.
    angle = vecA.angle(vecY);
    angleY.setText("" +
prepareForDisplay(angle));
```

```
the //Compute and display the angle relative to
    // z-axis.
    angle = vecA.angle(vecZ);
    angleZ.setText("" +
prepareForDisplay(angle));
```

```
the //Compute and display the angle relative to
    // XY plane
    angle = vecA.angle(vecXY);
    angleXY.setText("" +
prepareForDisplay(angle));
```

```
the //Compute and display the angle relative to
    // YZ plane
    angle = vecA.angle(vecYZ);
    angleYZ.setText("" +
prepareForDisplay(angle));
```

```
    //Compute and display the angle relative to
```

the

```
// ZX plane  
angle = vecA.angle(vecZX);  
angleZX.setText("" +  
prepareForDisplay(angle));
```

```
//Compute and display the angle of the  
projection onto  
// the XY plane relative to the x-axis  
angle = vecXY.angle(vecX);  
anglePX.setText("" +  
prepareForDisplay(angle));
```

```
//Compute and display the angle of the  
projection onto  
// the YZ plane relative to the y-axis  
angle = vecYZ.angle(vecY);  
anglePY.setText("" +  
prepareForDisplay(angle));
```

```
//Compute and display the angle of the  
projection onto  
// the ZX plane relative to the z-axis  
angle = vecZX.angle(vecZ);  
anglePZ.setText("" +  
prepareForDisplay(angle));
```

```
myCanvas.repaint();//Copy off-screen image to  
canvas.
```

```
}//end actionPerformed
```

Once the required vectors had been created in [Listing 3](#), there is nothing new in the code in [Listing 4](#). Therefore, no explanation of [Listing 4](#) should be required beyond the comments embedded in the code.

The bottom line on the program named DotProd3D05

The bottom line is that because the methods in the game-math library named **GM02** were designed to do most of the hard work, writing application programs such as **DotProd3D05** using the game-math library is not difficult at all. You simply need to understand how to organize your code to accomplish the things that you need to accomplish.

That concludes the explanation of the program named **DotProd3D05** .

The program named DotProd3D06

This program demonstrates how the dot product can be used to find vectors that are perpendicular to a given vector.

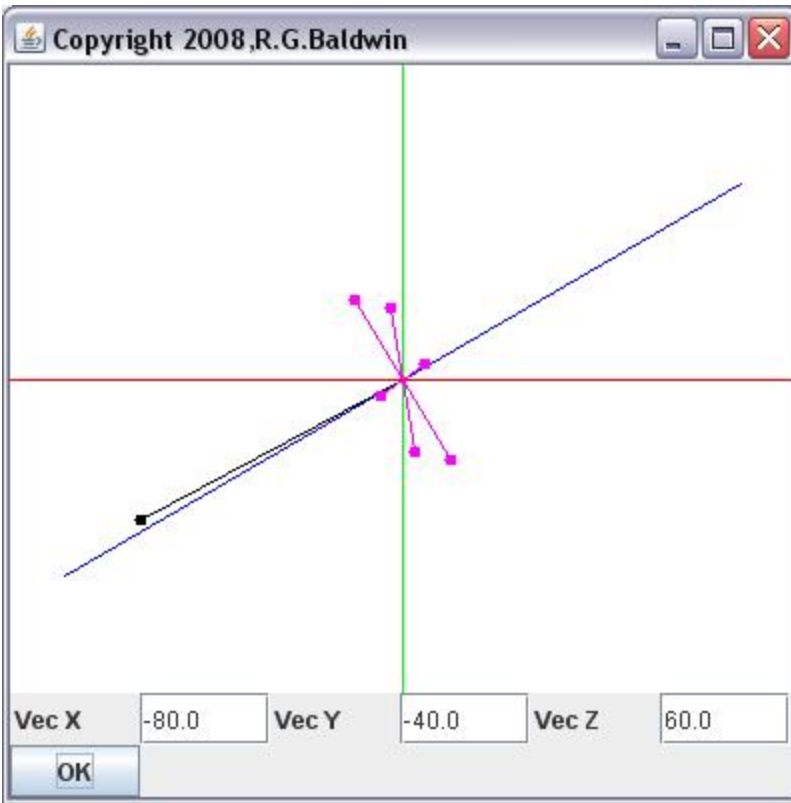
An infinite number of possibilities

Recall that you learned in the previous module that there are an infinite number of vectors that are perpendicular to a given vector in 3D. This program computes and displays normalized and scaled versions of six somewhat unique vectors of the infinite set of vectors that are perpendicular to a user specified vector.

The graphic output

The screen output from this program is shown in [Figure 4](#).

Figure 4 - Six (magenta) vectors that are perpendicular to a given (black) vector.



Output on the command-line screen

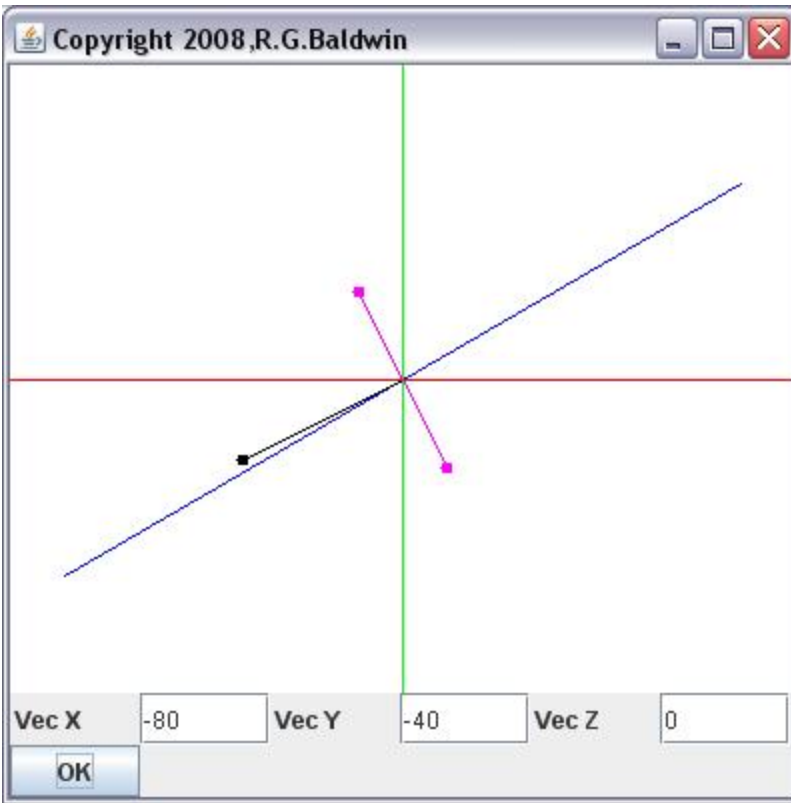
In addition to the graphic output shown in [Figure 4](#), the program also displays the values of three of the perpendicular vectors on the command-line screen along with the angle between the perpendicular vector and the user-specified vector. The angle should always be 90 degrees or very close to 90 degrees if the program is working properly. The other three of the six perpendicular vectors are simply negated versions of the three for which the values are displayed.

Special case of one zero coordinate value

If the user specifies one of the coordinate values to be zero (*or close to zero*), the program only computes and displays four of the possible vectors in order to avoid performing division by a near-zero value. In this case, the orientation of two of the vectors will overlay the orientation of the other two. Because the vectors are normalized to the same length and occupy the same space, you will only see two vectors instead of four. This is illustrated

in [Figure 5](#) where the value of the z-axis coordinate value was set to zero relative to the value used in [Figure 4](#).

Figure 5 - Screen output when one coordinate has a value of zero.



Special case of two coordinates with a value of zero

If the user specifies two of the coordinate values to be zero or close to zero, the program doesn't produce a valid result. Instead, it draws a perpendicular vector where all of the coordinate values are zero resulting in a magenta vector head at the origin. It also displays NaN (*Not a Number*) for the angle on the command line screen.

The graphical user interface

The GUI shown in [Figure 4](#) is provided to allow the user to enter three **double** values that define a **GM02.Vector3D** object. The GUI also provides an **OK** button in addition to a 3D drawing area.

When the user clicks the **OK** button, the program draws the user-specified vector in black with the tail located at the origin in 3D space. It draws normalized versions of the perpendicular vectors in magenta with their tails also located at the origin. Each normalized vector is scaled by a factor of 50 before it is drawn to cause it to be long enough to be seen.

A short review

Before getting into the programming details, we need to review some material from the previous module. Recall that I did some algebraic manipulations in the previous module and produced the equations shown in [Figure 6](#).

Figure 6 . A general formulation of 3D vector perpendicularity.

Figure 6 . A general formulation of 3D vector perpendicularity.

$$\text{dot product} = x1*x2 + y1*y2 + z1*z2$$

If the two vectors are perpendicular:

$$x1*x2 + y1*y2 + z1*z2 = 0.0$$

$$x1*x2 = -(y1*y2 + z1*z2)$$

$$x2 = -(y1*y2 + z1*z2)/x1$$

or

$$y2 = -(x1*x2 + z1*z2)/y1$$

or

$$z2 = -(x1*x2 + y1*y2)/z1$$

An infinite set of perpendicular vectors

The equations in [Figure 6](#) describe an infinite set of vectors that are all perpendicular to a given vector. Given these equations, and given the coordinates ($x1$, $y1$, and $z1$) of a vector for which we need to produce perpendicular vectors, we can assume values for any two of $x2$, $y2$, and $z2$. We can then determine the value for the other coordinate that will cause the new vector to be perpendicular to the given vector.

That is the procedure that is used by this program to produce three of the perpendicular vectors shown in [Figure 4](#). The remaining three perpendicular vectors shown in [Figure 4](#) are produced by negating the three vectors that are created using the procedure described above.

Beginning of the actionPerformed method

The only code in this program that is new to this module is contained in the **actionPerformed** method. This method is called to respond to a click on the **OK** button in [Figure 5](#). Therefore, I will confine my explanation to portions of the **actionPerformed** method. You can view a complete listing of this program in [Listing 11](#) near the end of the module.

The **actionPerformed** method begins in [Listing 5](#). Note that I deleted some of the code early in the method because it is very similar to code that I have explained before.

Listing 5 . Beginning of the actionPerformed method for the program named DotProd3D06.

```
public void actionPerformed(ActionEvent e){
```

Behavior of the actionPerformed method

The **actionPerformed** method contains three sections of code, each of which implements one of the equations in [Figure 6](#), to compute and draw one of the perpendicular vectors shown in [Figure 4](#). In addition, the code in the **actionPerformed** method draws the negative of those three vectors to produce the other three perpendicular vectors shown in [Figure 4](#).

Implement the last equation

[Listing 5](#) implements the last equation from [Figure 6](#), provided that the z-axis coordinate value for the given vector is greater than 0.001. As mentioned earlier, if the z-axis coordinate value is not greater than 0.001, the code in [Listing 5](#) is skipped and no effort is made to create and draw

that particular vector. This is done to prevent an attempt to divide by a zero or near-zero value.

A new ColMatrix3D object

[Listing 5](#) creates a new **ColMatrix3D** object with the x and y-axes coordinate values matching the corresponding values for the user specified vector. It executes the expression shown in [Listing 5](#) to compute the value of the z-axis coordinate that will cause the new vector to be perpendicular to the user-specified vector. *(The expression in [Listing 5](#) matches the last equation in [Figure 6](#).)*

A new Vector3D object

Then it uses the **ColMatrix3D** object described above to create, normalize, scale, and draw the first perpendicular vector. Following that, the code negates the perpendicular vector to create another perpendicular vector that points in the opposite direction.

Along the way, some information is displayed on the command-line screen.

The most important code

The most important code in [Listing 5](#), insofar as the objective of the program is concerned, is the expression that computes the z-axis coordinate value that will cause the new vector to be perpendicular to the user-specified vector.

Remainder of the actionPerformed method

[Listing 6](#) does essentially the same thing two more times to implement the other two equations shown in [Figure 6](#), creating and drawing four more perpendicular vectors in the process.

Listing 6 . Remainder of the actionPerformed method.

```
        if(Math.abs(yCoor) > 0.001){
            tempMatrix = new GM02.ColMatrix3D(
                xCoor, -(xCoor*xCoor +
zCoor*zCoor)/yCoor, zCoor);
            tempVec = new GM02.Vector3D(tempMatrix);
            System.out.println(tempVec);
            //Normalize and scale the perpendicular
vector.
            tempVec =
tempVec.normalize().scale(50.0);
            tempVec.draw(g2D, new GM02.Point3D(
                new
GM02.ColMatrix3D(0,0,0)));
            tempVec.negate().draw(g2D, new
GM02.Point3D(
                new
GM02.ColMatrix3D(0,0,0)));
            System.out.println(vecA.angle(tempVec));
        } //end if

        if(Math.abs(xCoor) > 0.001){
            tempMatrix = new GM02.ColMatrix3D(
                -(yCoor*yCoor + zCoor*zCoor)/xCoor,
yCoor, zCoor);
            tempVec = new GM02.Vector3D(tempMatrix);
            System.out.println(tempVec);
            //Normalize and scale the perpendicular
vector.
            tempVec =
tempVec.normalize().scale(50.0);
            tempVec.draw(g2D, new GM02.Point3D(
                new
GM02.ColMatrix3D(0,0,0)));
            tempVec.negate().draw(g2D, new
```

Listing 6 . Remainder of the actionPerformed method.

```
GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0));
    System.out.println(vecA.angle(tempVec));
} //end if

    myCanvas.repaint();//Copy off-screen image
to canvas.
    System.out.println();//blank line
} //end actionPerformed
//-----
-----//
```

[Listing 6](#) also causes the off-screen image to be displayed on the canvas.

That concludes the explanation of the program named **DotProd3D06** .

The program named DotProd3D04

The purpose of this program is to serve as a counterpoint to the program named **Prob3D03** , which demonstrates backface culling. (*I will explain the program named **Prob3D03** shortly.*)

This program draws the 3D object shown in [Figure 1](#), while the program named **DotProd3D03** draws the 3D object shown in [Figure 2](#). The difference between the two is that the object drawn by this program ([Figure 1](#)) does not incorporate *backface culling* to hide the lines on the back of the object. The object in [Figure 2](#) does incorporate backface culling. The difference is easy to see.

This program draws a 3D circular cylinder by stacking 20 circular disks on the x-z plane as shown in [Figure 1](#). The disks are centered on the y-axis

and are parallel to the x-z plane. The thickness of each disk is 5 vertical units. As mentioned above, there is no backface culling in this program, so all of the lines that should be hidden show through.

Will discuss in fragments

A complete listing of this program is provided in [Listing 12](#) near the end of the module. I will explain portions of the program using code fragments. However, I won't repeat explanations of code that I have already explained in this or in earlier modules.

The method named drawTheCylinder

All of the interesting code in this program is contained in the method named **drawTheCylinder**, which is shown in [Listing 7](#). *(Note that some of the code was deleted from [Listing 7](#) for brevity.)* This method is used to set the axes to the center of the off-screen image and to draw a 3D cylinder that is centered on the y-axis.

Listing 7 . The method named drawTheCylinder in DotProd3D04.

```
private void drawTheCylinder(Graphics2D g2D)
{
```

Behavior of the code

Basically, the code in [Listing 7](#) draws 21 circles, one above the other to represent the edges of the 20 circular disks. If you understand the trigonometry involved in drawing a circle, you should find the code in

[Listing 7](#) to be straightforward and no explanation beyond the embedded comments should be required. If you don't understand the trigonometry, you will simply have to take my word for it that the expressions in [Listing 7](#) are correct for drawing circles. A study of trigonometry is beyond the scope of this module.

The program named DotProd3D03

The purpose of this program is to demonstrate a practical use of the vector dot product -- backface culling. As before, the program draws a 3D circular cylinder by stacking 20 circular disks on the x-z plane. The disks are centered on the y-axis and are parallel to the x-z plane. The thickness of each disk is 5 vertical units.

Backface culling is incorporated using the dot product between a vector that is parallel to the viewpoint of the viewer and a vector that is perpendicular to the line being drawn to form the outline of a circle. The results are shown in [Figure 2](#).

Will discuss in fragments

A complete listing of this program is provided in [Listing 13](#) near the end of the module. I will explain portions of the program using code fragments. However, I won't repeat explanations of code that I have already explained in this or in earlier modules.

The method named drawTheCylinder

As before, all of the interesting code in this program is contained in the method named **drawTheCylinder**, which is shown in [Listing 8](#). (*Note that some of the code was deleted from [Listing 8](#) for brevity.*) This method is used to set the axes to the center of the off-screen image and to draw a 3D cylinder that is centered on the y-axis.

Listing 8 . The method named drawTheCylinder in DotProd3D03.

```
private void drawTheCylinder(Graphics2D g2D)
{
```

The new code and an exercise for the student

The new code in [Listing 8](#) is highlighted by a long explanatory comment near the middle of [Listing 8](#). I will leave it as an exercise for the student to think about the rationale that was used to decide which lines to draw and which lines to suppress depending on the algebraic sign of the dot product between the two vectors.

Otherwise, no explanation of the code should be necessary beyond the embedded comments.

Homework assignment

The homework assignment for this module was to study the Kjell tutorial through *Chapter 10, Angle between 3D Vectors*.

The homework assignment for the next module is to study the Kjell tutorial through *Chapter 11, Projections*.

In addition to studying the Kjell material, you should read at least the next two modules in this collection and bring your questions about that material to the next classroom session.

Finally, you should have begun studying the [physics material](#) at the beginning of the semester and you should continue studying one physics module per week thereafter. You should also feel free to bring your questions about that material to the classroom for discussion.

Run the program

I encourage you to copy the code from [Listing 9](#) through [Listing 13](#). Compile the code and execute it in conjunction with the game-math library named **GM02** in [Listing 9](#). Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this module, you learned how to apply the vector dot product to three different applications. You learned:

- how to use the dot product to compute nine different angles of interest that a vector makes with various elements in 3D space,
- you learned how to use the dot product to find six of the infinite set of vectors that are perpendicular to a given vector as shown in [Figure 4](#), and
- you learned how to use the dot product to perform back-face culling to convert the image in [Figure 1](#) to the image in [Figure 2](#).

What's next?

In the next module, you will learn about first-person computer games in general, and how to use the game-math library to write a first-person game in a 3D world using the game math library.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0150: Applications of the Vector Dot Product

- File: Game0150.htm
- Published: 10/23/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the programs discussed in this module are shown in [Listing 9](#) through [Listing 13](#) below.

Listing 9 . Source code for the game-math library named GM02.

```
/*GM02.java  
Copyright 2008, R.G.Baldwin  
Revised 02/08/08
```

This is an update to the game-math library named

THIS is an update to the game-math library named GM01.

The main purpose of this update was to add vector dot product and related capabilities to the library.

Please see the comments at the beginning of the library class named GM01 for a general description of the library.

The following methods are new instance methods of the indicated static top-level classes belonging to the class named GM02.

GM02.ColMatrix2D.dot - compute dot product of two ColMatrix2D objects.

GM02.Vector2D.dot - compute dot product of two Vector2D objects.

GM02.Vector2D.angle - compute angle between two Vector2D objects.

GM02.ColMatrix3D.dot - compute dot product of two ColMatrix3D objects

GM02.Vector3D.dot - compute dot product of two Vector3D objects.

GM02.Vector3D.angle - compute angle between two Vector3D objects.

Tested using JDK 1.6 under WinXP.

*****/

import java.awt.geom.*;

```

import java.awt.*;

public class GM02{
    //-----
    -----//

    //This method converts a ColMatrix3D object to a
    // ColMatrix2D object. The purpose is to accept
    // x, y, and z coordinate values and transform
    those
    // values into a pair of coordinate values
    suitable for
    // display in two dimensions.
    //See
    http://local.wasp.uwa.edu.au/~pbourke/geometry/
    // classification/ for technical background on
    the
    // transform from 3D to 2D.
    //The transform equations are:
    //  $x_{2d} = x_{3d} + z_{3d} * \cos(\theta)/\tan(\alpha)$ 
    //  $y_{2d} = y_{3d} + z_{3d} * \sin(\theta)/\tan(\alpha)$ ;
    //Let  $\theta = 30$  degrees and  $\alpha = 45$  degrees
    //Then: $\cos(\theta) = 0.866$ 
    //       $\sin(\theta) = 0.5$ 
    //       $\tan(\alpha) = 1$ ;
    //Note that the signs in the above equations
    depend
    // on the assumed directions of the angles as
    well as
    // the assumed positive directions of the axes.

    The
    // signs used in this method assume the
    following:
    //      Positive x is to the right.
    //      Positive y is up the screen.
    //      Positive z is protruding out the front of
    the

```

```

    //      screen.
    //      The viewing position is above the x axis
and to the
    //      right of the z-y plane.
    public static GM02.ColMatrix2D convert3Dto2D(
GM02.ColMatrix3D data){
    return new GM02.ColMatrix2D(
        data.getData(0) -
0.866*data.getData(2),
        data.getData(1) -
0.50*data.getData(2));
    }//end convert3Dto2D
    //-----
-----//

    //This method wraps around the translate method
of the
    // Graphics2D class. The purpose is to cause the
    // positive direction for the y-axis to be up
the screen
    // instead of down the screen. When you use this
method,
    // you should program as though the positive
direction
    // for the y-axis is up.
    public static void translate(Graphics2D g2D,
        double xOffset,
        double yOffset){
        //Flip the sign on the y-coordinate to change
the
        // direction of the positive y-axis to go up
the
        // screen.
        g2D.translate(xOffset, -yOffset);
    }//end translate
    //-----

```

```

-----//

    //This method wraps around the drawLine method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void drawLine(Graphics2D g2D,
                                double x1,
                                double y1,
                                double x2,
                                double y2){
        //Flip the sign on the y-coordinate value.
        g2D.drawLine((int)x1, -(int)y1, (int)x2, -
(int)y2);
    }//end drawLine
    //-----
-----//

```

```

    //This method wraps around the fillOval method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void fillOval(Graphics2D g2D,

```

```

        double x,
        double y,
        double width,
        double height){
    //Flip the sign on the y-coordinate value.
    g2D.fillOval((int)x,-(int)y,(int)width,
(int)height);
    }//end fillOval
    //-----
    -----//

```

```

    //This method wraps around the drawOval method
of the
    // Graphics class. The purpose is to cause the
positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void drawOval(Graphics2D g2D,
        double x,
        double y,
        double width,
        double height){
        //Flip the sign on the y-coordinate value.
        g2D.drawOval((int)x,-(int)y,(int)width,
(int)height);
    }//end drawOval
    //-----
    -----//

```

```

    //This method wraps around the fillRect method
of the
    // Graphics class. The purpose is to cause the

```

```

positive
    // direction for the y-axis to be up the screen
instead
    // of down the screen. When you use this method,
you
    // should program as though the positive
direction for
    // the y-axis is up.
    public static void fillRect(Graphics2D g2D,
                                double x,
                                double y,
                                double width,
                                double height){
        //Flip the sign on the y-coordinate value.
        g2D.fillRect((int)x,-(int)y,(int)width,
(int)height);
    }//end fillRect
    //-----
-----//

```

```

    //An object of this class represents a 2D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.
    public static class ColMatrix2D{
        double[] data = new double[2];

        public ColMatrix2D(double data0,double data1){
            data[0] = data0;
            data[1] = data1;
        }//end constructor
        //-----
-----//

```



```
        //Overridden toString method.
        public String toString(){
            return data[0] + "," + data[1];
        }//end overridden toString method
        //-----
    -----//
```

```
        public double getData(int index){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                return data[index];
            }//end else
        }//end getData method
        //-----
    -----//
```

```
        public void setData(int index,double data){
            if((index < 0) || (index > 1)){
                throw new IndexOutOfBoundsException();
            }else{
                this.data[index] = data;
            }//end else
        }//end setData method
        //-----
    -----//
```

```
        //This method overrides the equals method
        inherited
```

```
        // from the class named Object. It compares
        the values
        // stored in two matrices and returns true if
        the
        // values are equal or almost equal and
        returns false
        // otherwise.
```

```

        public boolean equals(Object obj){
            if(obj instanceof GM02.ColMatrix2D &&
Math.abs(((GM02.ColMatrix2D)obj).getData(0) -
                                                getData(0)) <=
0.00001 &&
Math.abs(((GM02.ColMatrix2D)obj).getData(1) -
                                                getData(1)) <=
0.00001){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
        -----//

        //Adds one ColMatrix2D object to another
        ColMatrix2D
        // object, returning a ColMatrix2D object.
        public GM02.ColMatrix2D add(GM02.ColMatrix2D
matrix){
            return new GM02.ColMatrix2D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1));
        }//end add

        //-----
        -----//

        //Subtracts one ColMatrix2D object from
        another
        // ColMatrix2D object, returning a ColMatrix2D
        object.

```

```

    // The object that is received as an incoming
    // parameter is subtracted from the object on
which
    // the method is called.
    public GM02.ColMatrix2D subtract(
                                GM02.ColMatrix2D
matrix){
    return new GM02.ColMatrix2D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1));
    }//end subtract
    //-----
-----//

    //Computes the dot product of two ColMatrix2D
    // objects and returns the result as type
double.
    public double dot(GM02.ColMatrix2D matrix){
    return getData(0) * matrix.getData(0)
        + getData(1) * matrix.getData(1);
    }//end dot
    //-----
-----//
    }//end class ColMatrix2D

//=====
====//

```

```

    //An object of this class represents a 3D column
matrix.
    // An object of this class is the fundamental
building
    // block for several of the other classes in the
    // library.

```

```

public static class ColMatrix3D{
    double[] data = new double[3];

    public ColMatrix3D(
        double data0,double data1,double
data2){
        data[0] = data0;
        data[1] = data1;
        data[2] = data2;
    }//end constructor
    //-----
    -----//

    public String toString(){
        return data[0] + "," + data[1] + "," +
data[2];
    }//end overridden toString method
    //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return data[index];
        }//end else
    }//end getData method
    //-----
    -----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            this.data[index] = data;
        }//end else
    }//end setData method

```

```

        //-----
        -----//

        //This method overrides the equals method
        inherited
        // from the class named Object. It compares
        the values
        // stored in two matrices and returns true if
        the
        // values are equal or almost equal and
        returns false
        // otherwise.
        public boolean equals(Object obj){
            if(obj instanceof GM02.ColMatrix3D &&
Math.abs(((GM02.ColMatrix3D)obj).getData(0) -
                                                getData(0)) <=
0.000001 &&
Math.abs(((GM02.ColMatrix3D)obj).getData(1) -
                                                getData(1)) <=
0.000001 &&
Math.abs(((GM02.ColMatrix3D)obj).getData(2) -
                                                getData(2)) <=
0.000001){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
        -----//

        //Adds one ColMatrix3D object to another
        ColMatrix3D

```

```

        // object, returning a ColMatrix3D object.
        public GM02.ColMatrix3D add(GM02.ColMatrix3D
matrix){
            return new GM02.ColMatrix3D(

getData(0)+matrix.getData(0),

getData(1)+matrix.getData(1),

getData(2)+matrix.getData(2));
        }//end add
        //-----
        -----//

        //Subtracts one ColMatrix3D object from
another
        // ColMatrix3D object, returning a ColMatrix3D
object.
        // The object that is received as an incoming
        // parameter is subtracted from the object on
which
        // the method is called.
        public GM02.ColMatrix3D subtract(
                                GM02.ColMatrix3D
matrix){
            return new GM02.ColMatrix3D(
                                getData(0)-
matrix.getData(0),
                                getData(1)-
matrix.getData(1),
                                getData(2)-
matrix.getData(2));
        }//end subtract
        //-----
        -----//

        //Computes the dot product of two ColMatrix3D

```

```

        // objects and returns the result as type
double.
        public double dot(GM02.ColMatrix3D matrix){
            return getData(0) * matrix.getData(0)
                + getData(1) * matrix.getData(1)
                + getData(2) * matrix.getData(2);
        }//end dot
        //-----
-----//
    }//end class ColMatrix3D

//=====
====//

//=====
====//

    public static class Point2D{
        GM02.ColMatrix2D point;

        public Point2D(GM02.ColMatrix2D point)
    { //constructor
        //Create and save a clone of the ColMatrix2D
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix2D
object
        // through use of its set method.
        this.point = new ColMatrix2D(
point.getData(0),point.getData(1));
    } //end constructor
    //-----

```



```

-
//-----
-----//

//Returns a reference to the ColMatrix2D
object that
// defines this Point2D object.
public GM02.ColMatrix2D getColMatrix(){
    return point;
} //end getColMatrix
//-----
-----//

//This method overrides the equals method
inherited
// from the class named Object. It compares
the values
// stored in the ColMatrix2D objects that
define two
// Point2D objects and returns true if they
are equal
// and false otherwise.
public boolean equals(Object obj){
    if(point.equals(((GM02.Point2D)obj).
getColMatrix())){
        return true;
    }else{
        return false;
    } //end else

} //end overridden equals method
//-----
-----//

//Gets a displacement vector from one Point2D
object

```

```

    // to a second Point2D object. The vector
points from
    // the object on which the method is called to
the
    // object passed as a parameter to the method.
Kjell
    // describes this as the distance you would
have to
    // walk along the x and then the y axes to get
from
    // the first point to the second point.
    public GM02.Vector2D getDisplacementVector(
                                                GM02.Point2D
point){
        return new GM02.Vector2D(new
GM02.ColMatrix2D(
                                point.getData(0)-
getData(0),
                                point.getData(1)-
getData(1)));
    }//end getDisplacementVector
    //-----
-----//

    //Adds a Vector2D to a Point2D producing a
    // new Point2D.
    public GM02.Point2D addVectorToPoint(
GM02.Vector2D vec){
        return new GM02.Point2D(new
GM02.ColMatrix2D(
                                getData(0) +
vec.getData(0),
                                getData(1) +
vec.getData(1)));
    }//end addVectorToPoint
    //-----

```

```

-----//

    //Returns a new Point2D object that is a clone
of
    // the object on which the method is called.
    public Point2D clone(){
        return new Point2D(
                                new
ColMatrix2D(getData(0),getData(1)));
    }//end clone
    //-----
-----//

    //The purpose of this method is to rotate a
point
    // around a specified anchor point in the x-y
plane.
    //The rotation angle is passed in as a double
value
    // in degrees with the positive angle of
rotation
    // being counter-clockwise.
    //This method does not modify the contents of
the
    // Point2D object on which the method is
called.
    // Rather, it uses the contents of that object
to
    // instantiate, rotate, and return a new
Point2D
    // object.
    //For simplicity, this method translates the
    // anchorPoint to the origin, rotates around
the
    // origin, and then translates back to the
    // anchorPoint.
    /*

```

See
<http://www.ia.hiof.no/~borres/cgraph/math/threed/p-threed.html> for a definition of the equations required to do the rotation.

```
x2 = x1*cos - y1*sin
y2 = x1*sin + y1*cos
*/
public GM02.Point2D rotate(GM02.Point2D
anchorPoint,
                        double angle){
    GM02.Point2D newPoint = this.clone();

    double tempX ;
    double tempY;

    //Translate anchorPoint to the origin
    GM02.Vector2D tempVec =
        new
GM02.Vector2D(anchorPoint.getColMatrix());
    newPoint =

newPoint.addVectorToPoint(tempVec.negate());

    //Rotate around the origin.
    tempX = newPoint.getData(0);
    tempY = newPoint.getData(1);
    newPoint.setData(new x coordinate
        0,

tempX*Math.cos(angle*Math.PI/180) -
tempY*Math.sin(angle*Math.PI/180));

    newPoint.setData(new y coordinate
        1,
```

```

tempX*Math.sin(angle*Math.PI/180) +
tempY*Math.cos(angle*Math.PI/180));

    //Translate back to anchorPoint
    newPoint =
newPoint.addVectorToPoint(tempVec);

    return newPoint;

} //end rotate
//-----
-----//

    //Multiplies this point by a scaling matrix
received
    // as an incoming parameter and returns the
scaled
    // point.
    public GM02.Point2D scale(GM02.ColMatrix2D
scale){
        return new GM02.Point2D(new ColMatrix2D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1)));
    } //end scale
    //-----
-----//

} //end class Point2D

//=====
====//

    public static class Point3D{

```

```

        GM02.ColMatrix3D point;

        public Point3D(GM02.ColMatrix3D point)
{
    //constructor
        //Create and save a clone of the ColMatrix3D
object
        // used to define the point to prevent the
point
        // from being corrupted by a later change in
the
        // values stored in the original ColMatrix3D
object
        // through use of its set method.
        this.point =
            new ColMatrix3D(point.getData(0),
                            point.getData(1),
                            point.getData(2));

    }
}

//-----
//-----

        public String toString(){
            return point.getData(0) + "," +
point.getData(1)
                                + "," +
point.getData(2);
        }
    }

//-----
//-----

        public double getData(int index){
            if((index < 0) || (index > 2)){
                throw new IndexOutOfBoundsException();
            }
            else{
                return point.getData(index);
            }
        }
    }
}

```

```

//-----
-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            point.setData(index,data);
        }//end else
    }//end setData
//-----
-----//

    //This method draws a small circle around the
location
    // of the point on the specified graphics
context.
    public void draw(Graphics2D g2D){

        //Get 2D projection coordinate values.
        ColMatrix2D temp = convert3Dto2D(point);
        drawOval(g2D,temp.getData(0)-3,
                    temp.getData(1)+3,
                    6,
                    6);
    }//end draw
//-----
-----//

    //Returns a reference to the ColMatrix3D
object that
    // defines this Point3D object.
    public GM02.ColMatrix3D getColMatrix(){
        return point;
    }//end getColMatrix
//-----
-----//

```

```

        //This method overrides the equals method
inherited
        // from the class named Object. It compares
the values
        // stored in the ColMatrix3D objects that
define two
        // Point3D objects and returns true if they
are equal
        // and false otherwise.
        public boolean equals(Object obj){
            if(point.equals(((GM02.Point3D)obj).
getColMatrix())){
                return true;
            }else{
                return false;
            }//end else

        }//end overridden equals method
        //-----
        -----//

        //Gets a displacement vector from one Point3D
object
        // to a second Point3D object. The vector
points from
        // the object on which the method is called to
the
        // object passed as a parameter to the method.
Kjell
        // describes this as the distance you would
have to
        // walk along the x and then the y axes to get
from
        // the first point to the second point.
        public GM02.Vector3D getDisplacementVector(

```



```

GM02.Point3D
point){
    return new GM02.Vector3D(new
GM02.ColMatrix3D(
        point.getData(0)-
getData(0),
        point.getData(1)-
getData(1),
        point.getData(2)-
getData(2)));
    }//end getDisplacementVector
    //-----
    -----//

    //Adds a Vector3D to a Point3D producing a
    // new Point3D.
    public GM02.Point3D addVectorToPoint(
GM02.Vector3D vec){
        return new GM02.Point3D(new
GM02.ColMatrix3D(
            getData(0) +
vec.getData(0),
            getData(1) +
vec.getData(1),
            getData(2) +
vec.getData(2)));
        }//end addVectorToPoint
        //-----
        -----//

        //Returns a new Point3D object that is a clone
of
        // the object on which the method is called.
        public Point3D clone(){
            return new Point3D(new
ColMatrix3D(getData(0),

```

```

getData(1),

getData(2)));
    }//end clone
    //-----
-----//

    //The purpose of this method is to rotate a
point
    // around a specified anchor point in the
following
    // order:
    // Rotate around z - rotation in x-y plane.
    // Rotate around x - rotation in y-z plane.
    // Rotate around y - rotation in x-z plane.
    //The rotation angles are passed in as double
values
    // in degrees (based on the right-hand rule)
in the
    // order given above, packaged in an object of
the
    // class GM02.ColMatrix3D. (Note that in this
case,
    // the ColMatrix3D object is simply a
convenient
    // container and it has no significance from a
matrix
    // viewpoint.)
    //The right-hand rule states that if you point
the
    // thumb of your right hand in the positive
direction
    // of an axis, the direction of positive
rotation
    // around that axis is given by the direction
that

```

```
// your fingers will be pointing.
//This method does not modify the contents of
the
// Point3D object on which the method is
called.
// Rather, it uses the contents of that object
to
// instantiate, rotate, and return a new
Point3D
// object.
//For simplicity, this method translates the
// anchorPoint to the origin, rotates around
the
// origin, and then translates back to the
// anchorPoint.
/*
See
http://www.ia.hiof.no/~borres/cgraph/math/threed/
p-threed.html for a definition of the
equations
required to do the rotation.
z-axis

$$\begin{aligned}x_2 &= x_1 \cos v - y_1 \sin v \\ y_2 &= x_1 \sin v + y_1 \cos v\end{aligned}$$

x-axis

$$\begin{aligned}y_2 &= y_1 \cos v - z_1 \sin v \\ z_2 &= y_1 \sin v + z_1 \cos v\end{aligned}$$

y-axis

$$\begin{aligned}x_2 &= x_1 \cos v + z_1 \sin v \\ z_2 &= -x_1 \sin v + z_1 \cos v\end{aligned}$$

*/
public GM02.Point3D rotate(GM02.Point3D
anchorPoint,
GM02.ColMatrix3D
angles){
```

```

    GM02.Point3D newPoint = this.clone();

    double tempX ;
    double tempY;
    double tempZ;

    //Translate anchorPoint to the origin
    GM02.Vector3D tempVec =
        new
    GM02.Vector3D(anchorPoint.getColMatrix());
    newPoint =

newPoint.addVectorToPoint(tempVec.negate());

    double zAngle = angles.getData(0);
    double xAngle = angles.getData(1);
    double yAngle = angles.getData(2);

    //Rotate around z-axis
    tempX = newPoint.getData(0);
    tempY = newPoint.getData(1);
    newPoint.setData(//new x coordinate
        0,

tempX*Math.cos(zAngle*Math.PI/180) -
tempY*Math.sin(zAngle*Math.PI/180));

        newPoint.setData(//new y coordinate
            1,

tempX*Math.sin(zAngle*Math.PI/180) +
tempY*Math.cos(zAngle*Math.PI/180));

    //Rotate around x-axis
    tempY = newPoint.getData(1);

```

```

        tempZ = newPoint.getData(2);
        newPoint.setData(//new y coordinate
                        1,

tempY*Math.cos(xAngle*Math.PI/180) -
tempZ*Math.sin(xAngle*Math.PI/180));

        newPoint.setData(//new z coordinate
                        2,

tempY*Math.sin(xAngle*Math.PI/180) +
tempZ*Math.cos(xAngle*Math.PI/180));

        //Rotate around y-axis
        tempX = newPoint.getData(0);
        tempZ = newPoint.getData(2);
        newPoint.setData(//new x coordinate
                        0,

tempX*Math.cos(yAngle*Math.PI/180) +
tempZ*Math.sin(yAngle*Math.PI/180));

        newPoint.setData(//new z coordinate
                        2,
                        -
tempX*Math.sin(yAngle*Math.PI/180) +

tempZ*Math.cos(yAngle*Math.PI/180));

        //Translate back to anchorPoint
        newPoint =
newPoint.addVectorToPoint(tempVec);

        return newPoint;

```

```

        }//end rotate
        //-----
-----//

        //Multiplies this point by a scaling matrix
        received
        // as an incoming parameter and returns the
        scaled
        // point.
        public GM02.Point3D scale(GM02.ColMatrix3D
scale){
        return new GM02.Point3D(new ColMatrix3D(
                                getData(0) *
scale.getData(0),
                                getData(1) *
scale.getData(1),
                                getData(2) *
scale.getData(2)));
        }//end scale
        //-----
-----//
    }//end class Point3D

//=====
====//

//=====
====//

    public static class Vector2D{
        GM02.ColMatrix2D vector;

        public Vector2D(GM02.ColMatrix2D vector)
    {//constructor
        //Create and save a clone of the ColMatrix2D

```

```

object
    // used to define the vector to prevent the
vector
    // from being corrupted by a later change in
the
    // values stored in the original ColVector2D
object.
    this.vector = new ColMatrix2D(
vector.getData(0),vector.getData(1));
    }//end constructor
    //-----
-----//

    public String toString(){
        return vector.getData(0) + "," +
vector.getData(1);
    }//end toString
    //-----
-----//

    public double getData(int index){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
    //-----
-----//

    public void setData(int index,double data){
        if((index < 0) || (index > 1)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else

```

```

        }//end setData
        //-----
    -----//

    //This method draws a vector on the specified
    graphics
    // context, with the tail of the vector
    located at a
    // specified point, and with a small filled
    circle at
    // the head.
    public void draw(Graphics2D g2D, GM02.Point2D
    tail){

        drawLine(g2D,
                    tail.getData(0),
                    tail.getData(1),
                    tail.getData(0)+vector.getData(0),
                    tail.getData(1)+vector.getData(1));

        fillOval(g2D,

tail.getData(0)+vector.getData(0)-3,

tail.getData(1)+vector.getData(1)+3,
                    6,
                    6);
    }//end draw
    //-----
    -----//

    //Returns a reference to the ColMatrix2D
    object that
    // defines this Vector2D object.
    public GM02.ColMatrix2D getColMatrix(){
        return vector;
    }//end getColMatrix

```



```

//-----
-----//

//This method overrides the equals method
inherited
// from the class named Object. It compares
the values
// stored in the ColMatrix2D objects that
define two
// Vector2D objects and returns true if they
are equal
// and false otherwise.
public boolean equals(Object obj){
    if(vector.equals((
(GM02.Vector2D)obj).getColMatrix())){
        return true;
    }else{
        return false;
    }//end else

} //end overridden equals method
//-----
-----//

//Adds this vector to a vector received as an
incoming
// parameter and returns the sum as a vector.
public GM02.Vector2D add(GM02.Vector2D vec){
    return new GM02.Vector2D(new ColMatrix2D(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1)));
} //end add
//-----
-----//

```

```

        //Returns the length of a Vector2D object.
        public double getLength(){
            return Math.sqrt(
                getData(0)*getData(0) +
getData(1)*getData(1));
        }//end getLength
        //-----
        -----//

        //Multiplies this vector by a scale factor
        received as
        // an incoming parameter and returns the
        scaled
        // vector.
        public GM02.Vector2D scale(Double factor){
            return new GM02.Vector2D(new ColMatrix2D(
                getData(0) *
factor,
                getData(1) *
factor));
        }//end scale
        //-----
        -----//

        //Changes the sign on each of the vector
        components
        // and returns the negated vector.
        public GM02.Vector2D negate(){
            return new GM02.Vector2D(new ColMatrix2D(
                -
getData(0),
                -
getData(1)));
        }//end negate
        //-----
        -----//

```

```

        //Returns a new vector that points in the same
        // direction but has a length of one unit.
        public GM02.Vector2D normalize(){
            double length = getLength();
            return new GM02.Vector2D(new ColMatrix2D(

getData(0)/length,

getData(1)/length));
        }//end normalize
        //-----
        -----//

        //Computes the dot product of two Vector2D
        // objects and returns the result as type
        double.
        public double dot(GM02.Vector2D vec){
            GM02.ColMatrix2D matrixA = getColMatrix();
            GM02.ColMatrix2D matrixB =
vec.getColMatrix();
            return matrixA.dot(matrixB);
        }//end dot
        //-----
        -----//

        //Computes and returns the angle between two
        Vector2D
        // objects. The angle is returned in degrees
        as type
        // double.
        public double angle(GM02.Vector2D vec){
            GM02.Vector2D normA = normalize();
            GM02.Vector2D normB = vec.normalize();
            double normDotProd = normA.dot(normB);
            return
Math.toDegrees(Math.acos(normDotProd));

```

```

        }//end angle
        //-----
-----//
    }//end class Vector2D

//=====
====//

    public static class Vector3D{
        GM02.ColMatrix3D vector;

        public Vector3D(GM02.ColMatrix3D vector)
    {//constructor
        //Create and save a clone of the ColMatrix3D
        object
        // used to define the vector to prevent the
        vector
        // from being corrupted by a later change in
        the
        // values stored in the original ColMatrix3D
        object.
        this.vector = new
        ColMatrix3D(vector.getData(0),
        vector.getData(1),
        vector.getData(2));
        }//end constructor
        //-----
-----//

        public String toString(){
            return vector.getData(0) + "," +
            vector.getData(1)
            + "," +
            vector.getData(2);
        }
    }
}

```

```

        }//end toString
        //-----
    -----//

    public double getData(int index){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            return vector.getData(index);
        }//end else
    }//end getData
    //-----
    -----//

    public void setData(int index,double data){
        if((index < 0) || (index > 2)){
            throw new IndexOutOfBoundsException();
        }else{
            vector.setData(index,data);
        }//end else
    }//end setData
    //-----
    -----//

    //This method draws a vector on the specified
    graphics
    // context, with the tail of the vector
    located at a
    // specified point, and with a small circle at
    the
    // head.
    public void draw(Graphics2D g2D,GM02.Point3D
    tail){

        //Get a 2D projection of the tail
        GM02.ColMatrix2D tail2D =
        convert3Dto2D(tail.point);

```

```

        //Get the 3D location of the head
        GM02.ColMatrix3D head =

tail.point.add(this.getColMatrix());

        //Get a 2D projection of the head
        GM02.ColMatrix2D head2D =
convert3Dto2D(head);
        drawLine(g2D,tail2D.getData(0),
                tail2D.getData(1),
                head2D.getData(0),
                head2D.getData(1));

        //Draw a small filled circle to identify the
head.
        fillOval(g2D,head2D.getData(0)-3,
                head2D.getData(1)+3,
                6,
                6);

        }//end draw
        //-----
-----//

        //Returns a reference to the ColMatrix3D
object that
        // defines this Vector3D object.
        public GM02.ColMatrix3D getColMatrix(){
            return vector;

        }//end getColMatrix
        //-----
-----//

        //This method overrides the equals method
inherited
        // from the class named Object. It compares

```

```

the values
    // stored in the ColMatrix3D objects that
define two
    // Vector3D objects and returns true if they
are equal
    // and false otherwise.
    public boolean equals(Object obj){
        if(vector.equals((
(GM02.Vector3D)obj).getColMatrix())){
            return true;
        }else{
            return false;
        }//end else

    }//end overridden equals method
    //-----
-----//

    //Adds this vector to a vector received as an
incoming
    // parameter and returns the sum as a vector.
    public GM02.Vector3D add(GM02.Vector3D vec){
        return new GM02.Vector3D(new ColMatrix3D(
vec.getData(0)+vector.getData(0),
vec.getData(1)+vector.getData(1),
vec.getData(2)+vector.getData(2)));
    }//end add
    //-----
-----//

    //Returns the length of a Vector3D object.
    public double getLength(){
        return Math.sqrt(getData(0)*getData(0) +

```

```

        `getData(1)*`getData(1) +
        `getData(2)*`getData(2));
    }//end getLength
    //-----
    -----//

    //Multiplies this vector by a scale factor
    received as
    // an incoming parameter and returns the
    scaled
    // vector.
    public GM02.Vector3D scale(Double factor){
        return new GM02.Vector3D(new ColMatrix3D(
            getData(0) *
factor,
            getData(1) *
factor,
            getData(2) *
factor));
    }//end scale
    //-----
    -----//

    //Changes the sign on each of the vector
    components
    // and returns the negated vector.
    public GM02.Vector3D negate(){
        return new GM02.Vector3D(new ColMatrix3D(
            -
getData(0),
            -
getData(1),
            -
getData(2)));
    }//end negate
    //-----
    -----//

```



```

        //Returns a new vector that points in the same
        // direction but has a length of one unit.
        public GM02.Vector3D normalize(){
            double length = getLength();
            return new GM02.Vector3D(new ColMatrix3D(

getData(0)/length,

getData(1)/length,

getData(2)/length));
        }//end normalize
        //-----
        -----//

        //Computes the dot product of two Vector3D
        // objects and returns the result as type
        double.
        public double dot(GM02.Vector3D vec){
            GM02.ColMatrix3D matrixA = getColMatrix();
            GM02.ColMatrix3D matrixB =
vec.getColMatrix();
            return matrixA.dot(matrixB);
        }//end dot
        //-----
        -----//

        //Computes and returns the angle between two
        Vector3D

        // objects. The angle is returned in degrees
        as type
        // double.
        public double angle(GM02.Vector3D vec){
            GM02.Vector3D normA = normalize();
            GM02.Vector3D normB = vec.normalize();
            double normDotProd = normA.dot(normB);

```

```

        return
Math.toDegrees(Math.acos(normDotProd));
    }//end angle
    //-----
-----//
    }//end class Vector3D

//=====
====//

//=====
====//

    //A line is defined by two points. One is called
the
    // tail and the other is called the head. Note
that this
    // class has the same name as one of the classes
in
    // the Graphics2D class. Therefore, if the class
from
    // the Graphics2D class is used in some future
upgrade
    // to this program, it will have to be fully
qualified.
    public static class Line2D{
        GM02.Point2D[] line = new GM02.Point2D[2];

        public Line2D(GM02.Point2D tail,GM02.Point2D
head){
            //Create and save clones of the points used
to
            // define the line to prevent the line from
being
            // corrupted by a later change in the
coordinate

```

```

        // values of the points.
        this.line[0] = new Point2D(new
GM02.ColMatrix2D(
tail.getData(0),tail.getData(1)));
        this.line[1] = new Point2D(new
GM02.ColMatrix2D(
head.getData(0),head.getData(1)));
    }//end constructor
    //-----
    -----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
            + line[0].getData(1) + "\nHead = "
            + line[1].getData(0) + ","
            + line[1].getData(1);
    }//end toString
    //-----
    -----//

    public GM02.Point2D getTail(){
        return line[0];
    }//end getTail
    //-----
    -----//

    public GM02.Point2D getHead(){
        return line[1];
    }//end getHead
    //-----
    -----//

    public void setTail(GM02.Point2D newPoint){
        //Create and save a clone of the new point
to

```

```

        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the
        // point.
        this.line[0] = new Point2D(new
GM02.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
    }//end setTail
    //-----
-----//

    public void setHead(GM02.Point2D newPoint){
        //Create and save a clone of the new point
to
        // prevent the line from being corrupted by
a
        // later change in the coordinate values of
the
        // point.
        this.line[1] = new Point2D(new
GM02.ColMatrix2D(
newPoint.getData(0),newPoint.getData(1)));
    }//end setHead
    //-----
-----//

    public void draw(Graphics2D g2D){
        drawLine(g2D, getTail().getData(0),
                    getTail().getData(1),
                    getHead().getData(0),
                    getHead().getData(1));
    }//end draw
    //-----
-----//

```

```

    }//end class Line2D

//=====
====//

    //A line is defined by two points. One is called
the
    // tail and the other is called the head.
    public static class Line3D{
        GM02.Point3D[] line = new GM02.Point3D[2];

        public Line3D(GM02.Point3D tail,GM02.Point3D
head){
            //Create and save clones of the points used
to
            // define the line to prevent the line from
being
            // corrupted by a later change in the
coordinate
            // values of the points.
            this.line[0] = new Point3D(new
GM02.ColMatrix3D(
tail.getData(0),
tail.getData(1),
tail.getData(2)));
            this.line[1] = new Point3D(new
GM02.ColMatrix3D(
head.getData(0),
head.getData(1),
head.getData(2)));

```

```

    }//end constructor
    //-----
    -----//

    public String toString(){
        return "Tail = " + line[0].getData(0) + ","
            + line[0].getData(1) + ","
            + line[0].getData(2)
            + "\nHead = "
            + line[1].getData(0) + ","
            + line[1].getData(1) + ","
            + line[1].getData(2);
    }//end toString
    //-----
    -----//

    public GM02.Point3D getTail(){
        return line[0];
    }//end getTail
    //-----
    -----//

    public GM02.Point3D getHead(){
        return line[1];
    }//end getHead
    //-----
    -----//

    public void setTail(GM02.Point3D newPoint){
        //Create and save a clone of the new point
to
        // prevent the line from being corrupted by
a
        // later change in the coordinate values of
the
        // point.
        this.line[0] = new Point3D(new

```

```

GM02.ColMatrix3D(
newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
    }//end setTail
    //-----
    -----//

    public void setHead(GM02.Point3D newPoint){
        //Create and save a clone of the new point
to        // prevent the line from being corrupted by
a        // later change in the coordinate values of
the        // point.
        this.line[1] = new Point3D(new
GM02.ColMatrix3D(
newPoint.getData(0),
newPoint.getData(1),
newPoint.getData(2)));
    }//end setHead
    //-----
    -----//

    public void draw(Graphics2D g2D){

        //Get 2D projection coordinates.
        GM02.ColMatrix2D tail =
convert3Dto2D(getTail().point);

```

```

        GM02.CoMatrix2D head =
convert3Dto2D(getHead().point);

        drawLine(g2D,tail.getData(0),
                    tail.getData(1),
                    head.getData(0),
                    head.getData(1));
    }//end draw
    //-----
-----//
} //end class Line3D

//=====
====//

} //end class GM02

```

Listing 10 . Source code for the program named DotProd3D05.

```

/*DotProd3D05.java
Copyright 2008, R.G.Baldwin
Revised 03/06/08

```

The purpose of this program is to demonstrate how the dot product can be used to compute nine different angles of interest that a vector makes with various elements in 3D space.

First, the program computes and displays the angle between a user-specified vector and each of the X, Y, and Z axes. These values are displayed with the labels Angle

X,
Angle Y, and Angle Z.

Then the program computes and displays the angle between

the vector and each of the XY, YZ, and ZX planes.
In

this case, the program computes the smallest possible angle by projecting the vector onto the plane and then

computing the angle between the vector and its projection.

These values are displayed with the labels Angle XY,
Angle YZ, and Angle ZX.

Finally, the program computes and displays the angle

between the projection of the vector on each of the three

planes and one of the axes that defines each plane.

Obviously, the angle between the projection and the other

axis that defines the plane is 90 degrees less the computed angle. Specifically the values that are computed

and displayed are:

Projection onto the XY plane relative to the x-axis,
displayed with the label Angle PX.

Projection onto the YZ plane relative to the y-axis,
displayed with the label Angle PY.

displayed with the label Angle PZ.

Projection onto the ZX plane relative to the z-axis,
displayed with the label Angle PZ.

All angles are reported as positive angles in degrees.

Study Kjell through Chapter 10, Angle between 3D Vectors.

A GUI is provided that allows the user to enter three double values that define a GM02.Vector3D object. The GUI also provides an OK button as well as nine text fields used to display the computed results described above.

In addition, the GUI provides a 3D drawing area.

When the user clicks the OK button, the program draws the user-specified vector in black with the tail located at the origin in 3D space. It also draws the projection of that vector in magenta on each of the XY, YZ, AND ZX planes

Tested using JDK 1.6 under WinXP.

```
*****  
*****/  
import java.awt.*;  
import javax.swing.*;
```

```

import java.awt.event.*;

class DotProd3D05{
    public static void main(String[] args){
        GUI guiObj = new GUI();

        }//end main
    }//end controlling class DotProd3D05
    //=====
    =====//

class GUI extends JFrame implements
ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.

    //User input components.
    JTextField vecX = new JTextField("50.0");
    JTextField vecY = new JTextField("100.0");
    JTextField vecZ = new JTextField("0.0");

    JTextField angleX = new JTextField("0");
    JTextField angleY = new JTextField("0");
    JTextField angleZ = new JTextField("0");

    JTextField angleXY = new JTextField("0");
    JTextField angleYZ = new JTextField("0");
    JTextField angleZX = new JTextField("0");

    JTextField analePX = new JTextField("0");

```

```

..... anglePY = new JTextField("0");
JTextField anglePZ = new JTextField("0");

JButton button = new JButton("OK");

//-----
-----//

GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Instantiate a JPanel that will house the
user input
    // components and set its layout manager.
    JPanel controlPanel = new JPanel();
    controlPanel.setLayout(new GridLayout(0,6));

    //Add the user input components and
appropriate labels
    // to the control panel.
    controlPanel.add(new JLabel(" Vec X "));
    controlPanel.add(vecX);

    controlPanel.add(new JLabel(" Vec Y "));
    controlPanel.add(vecY);

    controlPanel.add(new JLabel(" Vec Z "));
    controlPanel.add(vecZ);

    controlPanel.add(new JLabel(" Angle X "));
    controlPanel.add(angleX);

```

```

        controlPanel.add(new JLabel(" Angle Y "));
        controlPanel.add(angleY);

        controlPanel.add(new JLabel(" Angle Z "));
        controlPanel.add(angleZ);

        controlPanel.add(new JLabel(" Angle XY "));
        controlPanel.add(angleXY);

        controlPanel.add(new JLabel(" Angle YZ "));
        controlPanel.add(angleYZ);

        controlPanel.add(new JLabel(" Angle ZX "));
        controlPanel.add(angleZX);

        controlPanel.add(new JLabel(" Angle PX "));
        controlPanel.add(anglePX);

        controlPanel.add(new JLabel(" Angle PY "));
        controlPanel.add(anglePY);

        controlPanel.add(new JLabel(" Angle PZ "));
        controlPanel.add(anglePZ);

        controlPanel.add(button);

        //Add the control panel to the SOUTH position
in the
        // JFrame.
        this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

        //Create a new drawing canvas and add it to
the
        // CENTER of the JFrame above the control

```

```

        // CENTER OF THE FRAME ABOVE THE CENTER OF
panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(
BorderLayout.CENTER, myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
        osi = createImage(osiWidth, osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM02.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

        //Register this object as an action listener
on the
        // button.
        button.addActionListener(this);

        //Cause the overridden paint method belonging

```

```

to
    // myCanvas to be executed.
    myCanvas.repaint();

} //end constructor

//-----
-----//

//This method is used to draw orthogonal 3D axes
on the
// off-screen image that intersect at the
origin.
private void setCoordinateFrame(Graphics2D g2D){

    //Erase the screen
    g2D.setColor(Color.WHITE);
    GM02.fillRect(g2D, -osiWidth/2, osiHeight/2,

osiWidth, osiHeight);

    //Draw x-axis in RED
    g2D.setColor(Color.RED);
    GM02.Point3D pointA = new GM02.Point3D(
        new GM02.ColMatrix3D(-
osiWidth/2, 0, 0));
    GM02.Point3D pointB = new GM02.Point3D(
        new
GM02.ColMatrix3D(osiWidth/2, 0, 0));
    new GM02.Line3D(pointA, pointB).draw(g2D);

    //Draw y-axis in GREEN
    g2D.setColor(Color.GREEN);
    pointA = new GM02.Point3D(
        new GM02.ColMatrix3D(0, -
osiHeight/2, 0));
    pointB = new GM02.Point3D(
        new

```

```

GM02.ColMatrix3D(0,osiHeight/2,0));
    new GM02.Line3D(pointA,pointB).draw(g2D);

    //Draw z-axis in BLUE. Make its length the
    same as the

    // length of the x-axis.
    g2D.setColor(Color.BLUE);
    pointA = new GM02.Point3D(
        new GM02.ColMatrix3D(0,0,-
osiWidth/2));
    pointB = new GM02.Point3D(
        new
GM02.ColMatrix3D(0,0,osiWidth/2));
    new GM02.Line3D(pointA,pointB).draw(g2D);

} //end setCoordinateFrame method
//-----
-----//

//This method is called to respond to a click on
the
// button.
public void actionPerformed(ActionEvent e){

    //Erase the off-screen image and draw the
    axes.
    setCoordinateFrame(g2D);

    //Create one ColMatrix3D object based on the
    user
    // input values.
    GM02.ColMatrix3D matrixA = new
GM02.ColMatrix3D(

Double.parseDouble(vecX.getText()),

Double.parseDouble(vecY.getText()).

```



```
GM02.ColMatrix3D matrixZX = new
GM02.ColMatrix3D(
Double.parseDouble(vecX.getText()),
    0,
Double.parseDouble(vecZ.getText()));
```

```
//Use the ColMatrix3D objects to create
Vector3D
// objects representing the user-specified
vector and
// each of the axes.
GM02.Vector3D vecA = new
GM02.Vector3D(matrixA);
GM02.Vector3D vecX = new
GM02.Vector3D(matrixX);
GM02.Vector3D vecY = new
GM02.Vector3D(matrixY);
GM02.Vector3D vecZ = new
GM02.Vector3D(matrixZ);

//Create Vector3D objects that represent the
// projection of the user-specified vector on
each of
// the planes.
GM02.Vector3D vecXY = new
GM02.Vector3D(matrixXY);
GM02.Vector3D vecYZ = new
GM02.Vector3D(matrixYZ);
GM02.Vector3D vecZX = new
GM02.Vector3D(matrixZX);

//Draw the projection of the user specified
vector on
```

```

// each of the three planes.
g2D.setColor(Color.MAGENTA);
vecXY.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
vecYZ.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
vecZX.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));

//Draw the user-specified vector with its tail
at the
// origin.
g2D.setColor(Color.BLACK);
vecA.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));

//Compute and display the angle relative to
the
// x-axis.
double angle = vecA.angle(vecX);
angleX.setText("" +
prepareForDisplay(angle));

//Compute and display the angle relative to
the
// y-axis.
angle = vecA.angle(vecY);
angleY.setText("" +
prepareForDisplay(angle));

//Compute and display the angle relative to

```

```

//Compute and display the angle relative to
the
// z-axis.
angle = vecA.angle(vecZ);
angleZ.setText("" +
prepareForDisplay(angle));

```

```

//Compute and display the angle relative to
the
// XY plane
angle = vecA.angle(vecXY);
angleXY.setText("" +
prepareForDisplay(angle));

```

```

//Compute and display the angle relative to
the
// YZ plane
angle = vecA.angle(vecYZ);
angleYZ.setText("" +
prepareForDisplay(angle));

```

```

//Compute and display the angle relative to
the
// ZX plane
angle = vecA.angle(vecZX);
angleZX.setText("" +
prepareForDisplay(angle));

```

```

//Compute and display the angle of the
projection onto
// the XY plane relative to the x-axis
angle = vecXY.angle(vecX);
anglePX.setText("" +
prepareForDisplay(angle));

```

```

//Compute and display the angle of the

```

```

        //Compute and display the angle of the
projection onto
        // the YZ plane relative to the y-axis
        angle = vecYZ.angle(vecY);
        anglePY.setText("" +
prepareForDisplay(angle));

        //Compute and display the angle of the
projection onto
        // the ZX plane relative to the z-axis
        angle = vecZX.angle(vecZ);
        anglePZ.setText("" +
prepareForDisplay(angle));

        myCanvas.repaint();//Copy off-screen image to
canvas.

    }//end actionPerformed
    //-----
    -----//

    //The code in this method prepares a double
value for
    // display in a text field by eliminating
exponential
    // format for very small values and setting the
number
    // of decimal digits to four.
    private double prepareForDisplay(double data){
        //Eliminate exponential notation in the
display.
        if(Math.abs(data) < 0.001){
            data = 0.0;
        }//end if

        //Convert to four decimal digits.

```

```

        return ((int)(10000*data))/10000.0;
    }//end prepareForDisplay

//=====
====//

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the
        // screen or when the repaint method is called
on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI

```

Listing 11 . Source code for the program named DotProd3D06.

```

/*DotProd3D06.java
Copyright 2008, R.G.Baldwin
Revised 03/09/08

```

This program demonstrates how the dot product can be used to find vectors that are perpendicular to a given

vector.

The program computes and displays normalized and scaled versions of six of the infinite set of vectors that are perpendicular to a user specified vector.

If the user specifies one of the coordinates to be zero or close to zero, the program only computes and displays four of the possible vectors in order to avoid performing division by a near-zero value. For a value of zero, the orientation of two of the vectors will overlay the orientation of the other two. Because they are the same length, and occupy the same space, you will only see two vectors.

If the user specifies two of the coordinates to be zero or close to zero, the program doesn't produce a valid result. Instead, it displays the coordinates for a perpendicular vector where all of the coordinates are zero and displays NaN for the angle.

Study Kjell through Chapter 10, Angle between 3D Vectors.

A GUI is provided that allows the user to enter

three
double values that define a GM02.Vector3D object.
The GUI
also provides an OK button.

In addition, the GUI provides a 3D drawing area.

When the user clicks the OK button, the program
draws the
user-specified vector in black with the tail
located at
the origin in 3D space. It also draws normalized
versions
of the perpendicular vectors in magenta with their
tails
located at the origin. Each normalized vector is
scaled
by a factor of 50 before it is drawn.

The program also displays the values of three of
the
perpendicular vectors on the command-line screen
along
with the angle between the perpendicular vector
and the
user-specified vector. The angle should be 90
degrees or
at least very close to 90 degrees. The other three
perpendicular vectors are simply negated versions
of the
three for which the values are displayed.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/  
import java.awt.*;  
import javax.swing.*;
```



```

import java.awt.event.*;

class DotProd3D06{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
} //end controlling class DotProd3D06
//=====
=====//

class GUI extends JFrame implements
ActionListener{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 400;
    int vSize = 400;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.

    //User input components.
    JTextField vecX = new JTextField("50.0");
    JTextField vecY = new JTextField("50.0");
    JTextField vecZ = new JTextField("50.0");

    JButton button = new JButton("OK");

    //-----
    -----//

    GUI(){//constructor

        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
    }
}

```

```

        setTitle("Copyright 2008, R.G. Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Instantiate a JPanel that will house the
user input
// components and set its layout manager.
JPanel controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(0,6));

//Add the user input components and
appropriate labels
// to the control panel.
controlPanel.add(new JLabel(" Vec X "));
controlPanel.add(vecX);

controlPanel.add(new JLabel(" Vec Y "));
controlPanel.add(vecY);

controlPanel.add(new JLabel(" Vec Z "));
controlPanel.add(vecZ);

controlPanel.add(button);

//Add the control panel to the SOUTH position
in the
// JFrame.
this.getContentPane().add(
BorderLayout.SOUTH,controlPanel);

//Create a new drawing canvas and add it to
the
// CENTER of the JFrame above the control
panel.
myCanvas = new MyCanvas();

```

```

        this.getContentPane().add(
BorderLayout.CENTER, myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
osiWidth = myCanvas.getWidth();
osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
osi = createImage(osiWidth, osiHeight);
g2D = (Graphics2D)(osi.getGraphics());

        //Translate the origin to the center.

GM02.translate(g2D, 0.5*osiWidth, -0.5*osiHeight);

        //Register this object as an action listener
on the
        // button.
button.addActionListener(this);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.

```

```

        myCanvas.repaint();

    }//end constructor
    //-----
    -----//

    //This method is used to draw orthogonal 3D axes
    on the
    // off-screen image that intersect at the
    origin.
    private void setCoordinateFrame(Graphics2D g2D){

        //Erase the screen
        g2D.setColor(Color.WHITE);
        GM02.fillRect(g2D, -osiWidth/2, osiHeight/2,

osiWidth, osiHeight);

        //Draw x-axis in RED
        g2D.setColor(Color.RED);
        GM02.Point3D pointA = new GM02.Point3D(
            new GM02.ColMatrix3D(-
osiWidth/2, 0, 0));
        GM02.Point3D pointB = new GM02.Point3D(
            new
GM02.ColMatrix3D(osiWidth/2, 0, 0));
        new GM02.Line3D(pointA, pointB).draw(g2D);

        //Draw y-axis in GREEN
        g2D.setColor(Color.GREEN);
        pointA = new GM02.Point3D(
            new GM02.ColMatrix3D(0, -
osiHeight/2, 0));
        pointB = new GM02.Point3D(
            new
GM02.ColMatrix3D(0, osiHeight/2, 0));
        new GM02.Line3D(pointA, pointB).draw(q2D);
    }

```

```

        //Draw z-axis in BLUE. Make its length the
same as the
        // length of the x-axis.
        g2D.setColor(Color.BLUE);
        pointA = new GM02.Point3D(
                                new GM02.ColMatrix3D(0,0,-
osiWidth/2));
        pointB = new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,osiWidth/2));
        new GM02.Line3D(pointA,pointB).draw(g2D);

    }//end setCoordinateFrame method
    //-----
    -----//

    //This method is called to respond to a click on
the
    // button.
    public void actionPerformed(ActionEvent e){

        //Erase the off-screen image and draw the
axes.
        setCoordinateFrame(g2D);

        //Get and save the user specified coordinate
values.
        double xCoor =
Double.parseDouble(vecX.getText());
        double yCoor =
Double.parseDouble(vecY.getText());
        double zCoor =
Double.parseDouble(vecZ.getText());

        //Create a ColMatrix3D object based on the
user input

```

```

        // values.
        GM02.ColMatrix3D matrixA =
            new
GM02.ColMatrix3D(xCoor,yCoor,zCoor);

        //Use the ColMatrix3D object to create a
Vector3D
        // object representing the user-specified
vector.
        GM02.Vector3D vecA = new
GM02.Vector3D(matrixA);

        //Draw the user-specified vector with its tail
at the
        // origin.
        g2D.setColor(Color.BLACK);
        vecA.draw(g2D,new GM02.Point3D(
            new
GM02.ColMatrix3D(0,0,0)));

        //Create and draw the perpendicular vectors.
However,
        // if a coordinate value is near zero, don't
attempt
        // to create and draw the perpendicular vector
that
        // would require division by the near-zero
value.
        GM02.Vector3D tempVec;
        GM02.ColMatrix3D tempMatrix;
        g2D.setColor(Color.MAGENTA);

        if(Math.abs(zCoor) > 0.001){
            tempMatrix = new GM02.ColMatrix3D(
                xCoor,yCoor,-(xCoor*xCoor +
yCoor*yCoor)/zCoor);
            tempVec = new GM02.Vector3D(tempMatrix);

```

```

        System.out.println(tempVec);
        //Normalize and scale the perpendicular
vector.
        tempVec = tempVec.normalize().scale(50.0);
        tempVec.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
        tempVec.negate().draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
        System.out.println(vecA.angle(tempVec));
    }//end if

    if(Math.abs(yCoor) > 0.001){
        tempMatrix = new GM02.ColMatrix3D(
            xCoor,-(xCoor*xCoor +
zCoor*zCoor)/yCoor,zCoor);
        tempVec = new GM02.Vector3D(tempMatrix);
        System.out.println(tempVec);
        //Normalize and scale the perpendicular
vector.
        tempVec = tempVec.normalize().scale(50.0);
        tempVec.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
        tempVec.negate().draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
        System.out.println(vecA.angle(tempVec));
    }//end if

    if(Math.abs(xCoor) > 0.001){
        tempMatrix = new GM02.ColMatrix3D(
            -(yCoor*yCoor + zCoor*zCoor)/xCoor,
yCoor,zCoor);
        tempVec = new GM02.Vector3D(tempMatrix);
        System.out.println(tempVec);
    }
}

```

```

        //Normalize and scale the perpendicular
vector.
        tempVec = tempVec.normalize().scale(50.0);
        tempVec.draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
        tempVec.negate().draw(g2D,new GM02.Point3D(
                                new
GM02.ColMatrix3D(0,0,0)));
        System.out.println(vecA.angle(tempVec));
    }//end if

    myCanvas.repaint();//Copy off-screen image to
canvas.
    System.out.println();//blank line
} //end actionPerformed
//-----
-----//

//=====
=====//

//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method
will be
    // called when the JFrame and the Canvas
appear on the
    // screen or when the repaint method is called
on the
    // Canvas object.

    //The purpose of this method is to display the
// off-screen image on the screen.
    public void paint(Graphics g){
        g.drawImage(osi,0,0,this);
    }
}

```



```

        }//end overridden paint()
    }//end inner class MyCanvas

}//end class GUI

```

Listing 12 . Source code for the program named DotProb3D04.

```

/*DotProd3D04.java
Copyright 2008, R.G.Baldwin
Revised 03/07/08

```

The purpose of this program is serve as a counterpoint to the program named Prob3D03, which demonstrates backface culling.

This program draws the same 3D object as the one drawn in DotProd3D03 but without the benefit of backface culling.

Study Kjell through Chapter 10, Angle between 3D Vectors.

The program draws a 3D circular cylinder by stacking 20 circular disks on the x-z plane. The disks are centered on the y-axis and are parallel to the x-z plane. The thickness of each disk is 5 vertical units.

There is no backface culling in this program, so all of the lines that should be hidden show through.

Tested using JDK 1.6 under WinXP.

*****/

```
import java.awt.*;
import javax.swing.*;
```

```
class DotProd3D04{
    public static void main(String[] args){
        GUI guiObj = new GUI();
    }//end main
}//end controlling class DotProd3D04
//=====
=====//
```

```
class GUI extends JFrame{
    //Specify the horizontal and vertical size of a
    JFrame
    // object.
    int hSize = 230;
    int vSize = 250;
    Image osi;//an off-screen image
    int osiWidth;//off-screen image width
    int osiHeight;//off-screen image height
    MyCanvas myCanvas;//a subclass of Canvas
    Graphics2D g2D;//off-screen graphics context.
    //-----
    -----//
```

```
    GUI(){//constructor

        //Set JFrame size, title, and close operation.
        setSize(hSize,vSize);
        setTitle("Copyright 2008,R.G.Baldwin");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        //Create a new drawing canvas and add it to
the
        // CENTER of the JFrame above the control
panel.
        myCanvas = new MyCanvas();
        this.getContentPane().add(

BorderLayout.CENTER,myCanvas);

        //This object must be visible before you can
get an
        // off-screen image. It must also be visible
before
        // you can compute the size of the canvas.
        setVisible(true);

        //Make the size of the off-screen image match
the
        // size of the canvas.
        osiWidth = myCanvas.getWidth();
        osiHeight = myCanvas.getHeight();

        //Create an off-screen image and get a
graphics
        // context on it.
        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Call a method that sets the axes and draws
the
        // cylinder.
        drawTheCylinder(g2D);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.
```

```

        myCanvas.repaint();

    }//end constructor
    //-----
    -----//

    //This method is used to set the axes to the
    center of
    // the off-screen image and to draw a 3D
    cylinder that
    // is centered on the y-axis.
    private void drawTheCylinder(Graphics2D g2D){

        //Translate the origin to the center of the
        off-screen
        // image.

        GM02.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

        //Erase the screen
        g2D.setColor(Color.WHITE);
        GM02.fillRect(g2D,-osiWidth/2,osiHeight/2,

osiWidth,osiHeight);
        //Draw a cylinder by stacking 20 circular
        disks on
        // the x-z plane. The disks are centered on
        the
        // y-axis and are parallel to the x-z plane.
        The
        // thickness of each disk is 5 vertical units.
        GM02.Point3D tempPointA;
        GM02.Point3D tempPointB;
        g2D.setColor(Color.BLACK);

        for(int y = 0;y < 105;y += 5){//iterate on
        disks

```

```

        //Define the starting point on the circle
for this
        // disk.
        tempPointA = new GM02.Point3D(
            new GM02.ColMatrix3D(76,y -
osiHeight/4,0));
        //Iterate on points on the circle that
represents
        // this disk.
        for(int cnt = 0;cnt < 360;cnt++){//360
points
            //Compute the next point on the circle.
            tempPointB =
                new GM02.Point3D(new GM02.ColMatrix3D(
76*Math.cos(Math.toRadians(cnt*360/360)),
                y - osiHeight/4,
76*Math.sin(Math.toRadians(cnt*360/360))));

            //Draw the line in 3D. Note that there is
no
            // backface culling in this program.
            new
GM02.Line3D(tempPointA,tempPointB).draw(g2D);

            //Save the point for use in drawing the
next line.
            tempPointA = tempPointB;
        }//end for loop on points
    }//end for loop on disks

} //end drawTheCylinder method

//=====
====//

```

```

    //This is an inner class of the GUI class.
    class MyCanvas extends Canvas{
        //Override the paint() method. This method
will be
        // called when the JFrame and the Canvas
appear on the
        // screen or when the repaint method is called
on the
        // Canvas object.
        //The purpose of this method is to display the
        // off-screen image on the screen.
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI

```

Listing 13 . Source code for the program named DotProb3D03.

```

/*DotProd3D03.java
Copyright 2008, R.G.Baldwin
Revised 03/07/08

```

The purpose of this program is to demonstrate a practical use of the vector dot product - backface culling.

Study Kjell through Chapter 10, Angle between 3D Vectors.

The program draws a 3D circular cylinder by stacking 20 circular disks on the x-z plane. The disks are centered on the y-axis and are parallel to the x-z plane. The

thickness of each disk is 5 vertical units.

Backface culling is done using the dot product between a

vector that is parallel to the viewpoint of the viewer and

a vector that is perpendicular to the line being drawn to

form the outline of a disk. The backface culling process is good but not perfect. There is some leakage

around the back on the right and left sides of the cylinder and some short lines segments are visible that

should not be visible.

Tested using JDK 1.6 under WinXP.

```
*****  
*****/
```

```
import java.awt.*;  
import javax.swing.*;
```

```
class DotProd3D03{  
    public static void main(String[] args){  
        GUI guiObj = new GUI();  
    }//end main
```

```
}//end controlling class DotProd3D03
```

```
//=====
```

```
class GUI extends JFrame{  
    //Specify the horizontal and vertical size of a  
    JFrame  
    // object.  
    int hSize = 230;  
    int vSize = 250;
```

```

Image osi;//an off-screen image
int osiWidth;//off-screen image width
int osiHeight;//off-screen image height
MyCanvas myCanvas;//a subclass of Canvas

Graphics2D g2D;//off-screen graphics context.
//-----
-----//

GUI(){//constructor

    //Set JFrame size, title, and close operation.
    setSize(hSize,vSize);
    setTitle("Copyright 2008,R.G.Baldwin");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Create a new drawing canvas and add it to
the
    // CENTER of the JFrame above the control
panel.
    myCanvas = new MyCanvas();
    this.getContentPane().add(
BorderLayout.CENTER,myCanvas);

    //This object must be visible before you can
get an
    // off-screen image. It must also be visible
before
    // you can compute the size of the canvas.
    setVisible(true);

    //Make the size of the off-screen image match
the
    // size of the canvas.
    osiWidth = myCanvas.getWidth();
    osiHeight = myCanvas.getHeight();

```



```

        //Create an off-screen image and get a
graphics
        // context on it.

        osi = createImage(osiWidth,osiHeight);
        g2D = (Graphics2D)(osi.getGraphics());

        //Call a method that sets the axes and draws
the
        // cylinder.
        drawTheCylinder(g2D);

        //Cause the overridden paint method belonging
to
        // myCanvas to be executed.
        myCanvas.repaint();

    }//end constructor
    //-----
    -----//

    //This method is used to set the axes to the
center of
    // the off-screen image and to draw a 3D
cylinder that
    // is centered on the y-axis.
    private void drawTheCylinder(Graphics2D g2D){

        //Translate the origin to the center of the
off-screen
        // image.

        GM02.translate(g2D,0.5*osiWidth,-0.5*osiHeight);

        //Erase the screen
        g2D.setColor(Color.WHITE);
        GM02.fillRect(g2D,-osiWidth/2,osiHeight/2,

```

```

osiWidth,osiHeight);

    //Get a vector that is approximately parallel
to the
    // viewpoint of the viewer. The best values
for this
    // vector were determined experimentally using
this
    // program and the earlier program named
DotProd3D02.
    GM02.Vector3D viewPoint =
        new GM02.Vector3D(new
GM02.ColMatrix3D(43,5,50));

    //Draw a cylinder by stacking 20 circular
disks on
    // the x-z plane. The disks are centered on
the
    // y-axis and are parallel to the x-z plane.
The
    // thickness of each disk is 5 vertical units.
GM02.Point3D tempPointA;
GM02.Point3D tempPointB;
g2D.setColor(Color.BLACK);

    for(int y = 0;y < 105;y += 5){//iterate on
disks
        //Define the starting point on the circle
for this
        // disk.
        tempPointA = new GM02.Point3D(new
GM02.ColMatrix3D(
                                76,y -
osiHeight/4,0));
        //Iterate on points on the circle that
represents

```

```

        // this disk.
        for(int cnt = 0;cnt < 360;cnt++){//360
points
            //Compute the next point on the circle.

            tempPointB =
                new GM02.Point3D(new GM02.ColMatrix3D(
76*Math.cos(Math.toRadians(cnt*360/360)),
                    y - osiHeight/4,

76*Math.sin(Math.toRadians(cnt*360/360))));

            //Do backface culling using the dot
product of the
            // viewpoint vector and a vector that is
almost
            // perpendicular to the line being drawn.
If the
            // dot product is negative, or if the disk
being
            // drawn is the top disk on the stack,
draw the
            // line. Otherwise, don't draw the line.
            //The perpendicular vector used in the dot
            // product is the displacement vector from
the
            // origin to a point that defines one end
of the
            // line being drawn. Note that this vector
is not
            // perfectly perpendicular to the line
being
            // drawn. Later in the series, we will
learn about
            // and use the cross product to get
perpendicular
            // vectors.

```

```

        if((tempPointB.getDisplacementVector(
                                new GM02.Point3D(
                                    new
GM02.ColMatrix3D(0,0,0)))
                                .dot(viewPoint)
< 0.0)
                                || (y
== 100)){
        //Draw the line in 3D.
        new GM02.Line3D(
tempPointA,tempPointB).draw(g2D);
        }//end if

        //Save the point for use in drawing the
next line.
        tempPointA = tempPointB;
    }//end for loop on points
} //end for loop on disks

} //end drawTheCylinder method

//=====
=====//

```

```


//This is an inner class of the GUI class.
class MyCanvas extends Canvas{
    //Override the paint() method. This method
will be
    // called when the JFrame and the Canvas
appear on the
    // screen or when the repaint method is called
on the
    // Canvas object.
    //The purpose of this method is to display the
    // off-screen image on the screen.

```

```
        public void paint(Graphics g){
            g.drawImage(osi,0,0,this);
        }//end overridden paint()

    }//end inner class MyCanvas

} //end class GUI
```



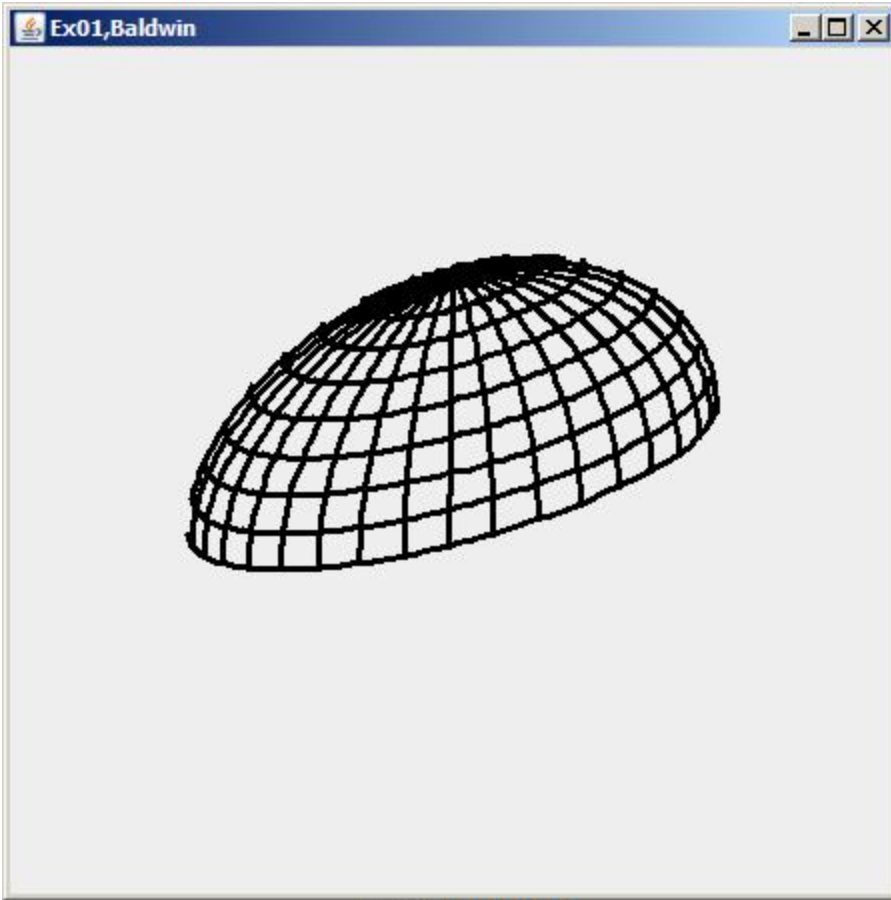
Exercises

Exercise 1

Using Java and the game-math library named **GM02** , or using a different programming environment of your choice, write a program that creates the drawing of half of a 3D sphere protruding upward from the x-z plane as shown in [Figure 7](#) . The north and south poles of the sphere lie on the y-axis, but only the northern hemisphere is visible.

Cause your name to appear in the screen output in some manner.

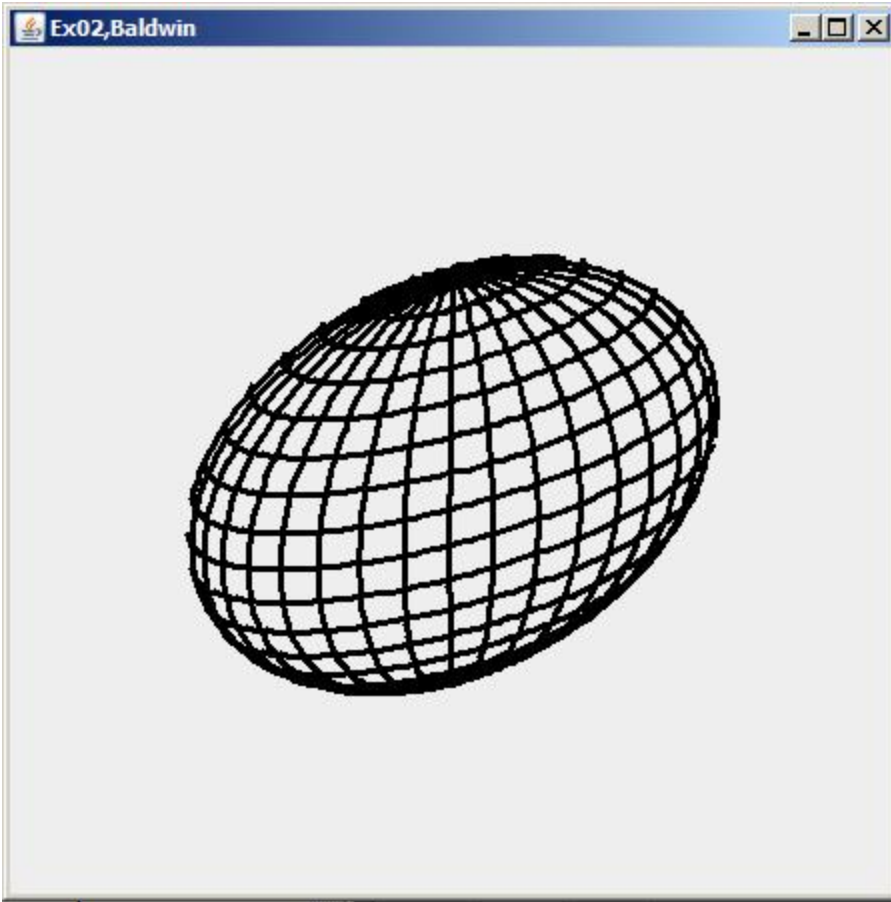
Figure 7 - Output from Exercise 1.



Exercise 2

Beginning with a program similar to the one that you wrote in [Exercise 1](#), create a drawing of a 3D sphere as shown in [Figure 8](#). The north and south poles of the sphere lie on the y-axis.

Figure 8 - Output from Exercise 2.



-end-

GAME 2302-0300 Introduction to Physics Component

This module is the first in a series of modules designed for teaching the physics component of GAME2302 Mathematical Applications for Game Development at Austin Community College in Austin, TX.

Table of Contents

- [Preface](#)
- [Miscellaneous](#)

Preface

This module is the first in a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0300 Introduction to Physics Component
- File: Game0300.htm
- Published: 10/11/12
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0310 JavaScript

This module provides an introductory JavaScript programming tutorial.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [Discussion](#)
 - [Why I chose JavaScript](#)
 - [Facilities required](#)
 - [A minimal JavaScript script](#)
 - [Strings](#)
 - [Screen output](#)
 - [Structured programming](#)
 - [Functions](#)
 - [Arithmetic operators](#)
 - [Sequence](#)
 - [Selection](#)
 - [Relational and logical operators](#)
 - [Variables](#)
 - [The string concatenation operator](#)
 - [Repetition](#)
 - [Programming errors](#)
 - [Assistance using Google Chrome](#)
 - [Assistance using Firefox](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Output from script in Listing 1.
- [Figure 2](#). Output from script in Listing 2.
- [Figure 3](#). Binary arithmetic operators.
- [Figure 4](#). General syntax for selection statement.
- [Figure 5](#). Real-world analogy of a selection statement.
- [Figure 6](#). Output from script in Listing 3.
- [Figure 7](#). Relational operators.
- [Figure 8](#). Output from script in Listing 4.
- [Figure 9](#). General syntax for a while loop.
- [Figure 10](#). Output from script in Listing 5.

Listings

- [Listing 1](#). A minimal JavaScript script.
- [Listing 2](#). An example function named getHalf.

- [Listing 3](#). A selection script example.
- [Listing 4](#). A sample script that uses variables.
- [Listing 5](#). A simple while loop.

Discussion

The goal of this module is to provide an introductory JavaScript programming tutorial for students with no programming experience.

Why I chose JavaScript

I chose JavaScript for use in these physics modules for several reasons.

Free

First, JavaScript is free. The capability to program in JavaScript is available to anyone who has a modern browser installed on their computer. Therefore, cost is not an excuse for not learning to program with JavaScript.

If you are reading this module using a modern browser, you have the ability to program using JavaScript immediately. You don't have to go out and buy anything, so that isn't an excuse for putting it off until tomorrow.

If you don't have a modern browser, you can download a free copy of the Firefox browser at <http://www.mozilla.com/en-US/firefox/firefox.html>.

Fun

Also, programming with JavaScript can be fun. There are a lot of really interesting things that you can do with JavaScript such as playing sound files (see <http://www.javascripiter.net/faq/sound/play.htm>).

OOP

JavaScript encompasses modern programming concepts. For example, JavaScript is based on the concept of objects. Object-Oriented

Programming (OOP) is here to stay. (For an extensive discussion of OOP, see the early lessons in my online programming tutorials at <http://www.dickbaldwin.com/toc.htm>.)

A free audible, tactile scientific calculator

I also chose JavaScript because you will be able to use it to create your own scientific calculator. You can use JavaScript to create solutions to many of the exercises in the modules in this collection of physics concepts.

Facilities required

To use JavaScript for its intended purpose in the modules in this collection, you will need the following:

- Access to the Internet.
- A modern browser such as Firefox 3.6 (see <http://www.mozilla.com/en-US/firefox/firefox.html>).
- A plain text editor such as Windows notepad, or my favorite, Arachnophilia, which can be downloaded for free at <http://www.arachnoid.com/arachnophilia/>.

A minimal JavaScript script

[Listing 1](#) shows a minimal JavaScript script.

Listing 1 . A minimal JavaScript script.

Listing 1 . A minimal JavaScript script.

```
<!-- File JavaScript01.html -->
<html><body>
<script language="JavaScript1.3">

document.write("Insert JavaScript between
script tags.", "<br/>")
document.write("Hello from JavaScript")

</script>
</body></html>
```

Run the script

To run this JavaScript script:

- Copy all of the text from the body of [Listing 1](#) into your plain text editor and save the file with an extension of .html (test.html for example).
- Open that file in your browser. (In most cases, you should be able to simply drag the file and drop it onto an open browser page to open it. If that doesn't work, open it from the browser's **File** menu.)

And the result is...

When you do that, the text shown in [Figure 1](#) should appear in the browser window.

Figure 1 . Output from script in Listing 1.

Figure 1 . Output from script in Listing 1.

```
Insert JavaScript between script tags.  
Hello from JavaScript
```

That's all there is to it

All you have to do to write and run a JavaScript script is to:

- Copy the text (often referred to as code or source code) from [Listing 1](#) into your plain text editor.
- Replace the code between the two lines containing the word **script** with the code for your new script, leaving the two lines containing the word **script** intact.
- Save the file with an extension of .html. (The file can have any legal file name so long as the extension is .html.)
- Open the new html file in your browser.

When you do that, the script code will be executed.

Display results in the browser window

If your script code contains statements that begin with **document.write** , followed by a pair of matching parentheses, (as shown in [Listing 1](#)), the code in the parentheses will be evaluated and the results of that evaluation will appear in your browser window.

At that point, you will have access to your script code as well as the results of running your script.

Strings

In programming, we often refer to a group of sequential characters (such as your first name) as a **string** . In JavaScript format (often called syntax),

such groups of characters are surrounded by matching quotation marks to cause the group to be recognized as a single string.

The following strings appear in the JavaScript code in [Listing 1](#):

1. "Insert JavaScript between script tags."
2. "
"
3. "Hello from JavaScript"

The first two strings appear, separated by a comma, inside the matching parentheses following the first occurrence of **document.write** . (We often call the items inside the matching parentheses the argument list.)

The last item in the above list appears in the argument list following the second occurrence of **document.write** in [Listing 1](#).

Screen output

If you examine [Figure 1](#), you will see that the first and third items from the [above list](#) appear, without their quotation marks, in the browser window. However, the second item does not appear in the browser window.

Purpose of document.write

Although this is a simplification, for purposes of the modules in this collection, it will suffice to say that the purpose of the **document.write** command is to cause the items in its argument list to be displayed in the browser window. You can place one or more items in the argument list. If there is more than one item in the argument list, the items must be separated by a comma as shown in [Listing 1](#).

Displaying strings

With some exceptions, items that appear in the argument list surrounded by matching quotation marks (strings) will be displayed in the browser window.

The exceptions include the second item in the [above list](#). This item is a special command to the browser, commonly known as a **break** tag. The occurrence of a **break** tag tells the browser to go down to the next line before displaying any additional material.

Such browser commands usually begin with a left angle bracket as shown in [Listing 1](#). Because of this, it is usually wise to avoid displaying strings that begin with left angle brackets unless you know for sure that your string won't be interpreted as a command to the browser.

Displaying material other than strings

I will show you how to display material other than strings in a later section titled [The string concatenation operator](#). Before getting into that, however, I will discuss several other topics.

Structured programming

When a student enrolls in the Object-Oriented Programming course that I teach at Austin Community College, I expect that student to have knowledge of something called structured programming, which is generally defined as including the following topics:

1. Functions with parameter passing
2. Sequence
3. Selection (if-else)
4. Repetition (for, while, and do-while loops)

I also expect them to know how to use variables and how to use operators (add, subtract, multiply, divide, etc.).

You will need to have an introductory knowledge of these topics to understand the JavaScript scripts that I will use to explain physics concepts in later modules. I will briefly explain these topics in this module and will discuss them further in later modules where they are used.

Functions

Functions, or procedures as they are called in some languages, provide a fundamental building block for virtually every programming language. The purpose of a function is to encapsulate the ability to perform a task into a single set of code and to be able to execute that code from a variety of different locations within the script. This can eliminate the requirement to repeat the same code over and over when the same task is required at multiple points in the script.

The surface area of a sphere

For example, if you write a script that frequently needs to calculate the surface area of a sphere, you can encapsulate those calculations in a function. Then, whenever your script needs to know the surface area of a sphere, it can simply call the function and provide the radius of the sphere as a parameter. The function will perform the calculation and return the answer to be used by the script at that point.

A JavaScript function definition

A JavaScript function definition has the following basic parts as shown in [Listing 2](#):

1. The **function** keyword.
2. The function name.
3. A comma-separated list of arguments enclosed in parentheses. (If there are no arguments, which is perfectly legal, the parentheses must still follow the function name but they will be empty.)
4. The statements in the body of the function enclosed in curly brackets.

Two sides to every function

There are two sides to the use of every function:

1. The function definition.
2. The function call.

The definition names the function and specifies how it will behave when it is called.

The call to the function temporarily passes control to the statements in the function causing them to behave as previously defined.

Once the statements have been executed, control is returned to the point in the script where the call was made. The function may, or may not return a value when it returns control.

The argument list

As in the sphere example discussed above, it is often useful to pass information to the function for it to use in doing whatever it is supposed to do (but this is not always required). When we call the function, we include parameters in the call to the function that match up with the argument list mentioned above. That is the mechanism used to pass information to a function. (This will probably make more sense when you see an example. Again, in some cases, no arguments are required.)

The purpose of a function

Usually (but not always), the purpose of a function is to calculate or otherwise determine some value and return it. In the sphere example mentioned earlier, the purpose of the function would be to calculate and return the surface area of the sphere. Returning a value is accomplished using the **return** keyword in the body of the function.

Sometimes, the purpose of a function is not to return a value, but instead to cause some action to occur, such as displaying information in the browser window. In that case, a **return** statement is not required. However, it doesn't cause any problem to put a **return** statement at the end of the function's body with nothing to the right of the word *return* .

An example function named getHalf

The code in [Listing 2](#) defines a function named **getHalf** and then calls that function from two different locations in a script, passing a different

parameter value with each call.

Listing 2 . An example function named getHalf.

```
<!-- File JavaScript02.html -->
<html><body>
<script language="JavaScript1.3">

//This is the syntax for a comment.

//Define the function named getHalf()
function getHalf(incomingParameter) {
    return incomingParameter/2;
}//end function getHalf()

document.write("Call getHalf for 10.6",
<br/>")
document.write("Half is: ", getHalf(10.6),
<br/>");

document.write("Call getHalf again for 12.3",
<br/>")
document.write("Half is: ", getHalf(12.3));

</script>
</body></html>
```

A note about comments

Note the line of code immediately following the first **script** tag that begins with `//`. Whenever JavaScript code contains such a pair of slash marks (that are not inside of a quoted string), everything from that point to the end of the line is treated as a comment. A comment is intended for human consumption only and is completely ignored when the script is run.

The function definition

The function definition in [Listing 1](#) consists of the three lines of code following the comment that begins with `///Define the function...`

As explained earlier, this function definition contains:

- The **function** keyword
- The function name: **getHalf**
- An argument list containing a single parameter named **incomingParameter**
- A function body enclosed in a pair of matching curly brackets

Simpler than the surface area of a sphere

The function named **getHalf** is somewhat simpler than one that could be used to calculate the surface area of a sphere, but the basic concept is the same. This function expects to receive one parameter.

The code in the body of the function uses the division operator, `/`, to divide the value of the incoming parameter by 2. Then it returns the result of that calculation. When the function returns, the return value will replace the call to the function in the calling script.

Two calls to the function

The function is called twice in the body of the script in [Listing 2](#), passing a different value for the parameter during each call.

Each call to the function named **getHalf** is embedded as one of the elements in the argument list following **document.write** .

write is also a function

Although I didn't mention this earlier because you weren't ready for it yet, **write** is also the name of a function. However, the **write** function is predefined in JavaScript and we can use it to display information in the browser window without having to define it first.

(Actually, **write** is a special kind of a function that we call a **method**, but you don't need to worry about that unless you want to dig much deeper into the object-oriented aspects of JavaScript.)

Two calls to the **getHalf** function

The script in [Listing 2](#) calls the function named **getHalf** twice in two different locations, each of which is embedded in the argument list of a call to the **write** method. Each call passes a different parameter value to the **getHalf** function.

The order of operations

When a call to a function or method (such as the call to the **write** method) includes a call to another function or method in its argument list, the call to the function in the argument list must be completed before the call can be made to the function having the argument list. Therefore in each of these two cases, the call to the **getHalf** function must be completed before the call to the **write** method can be executed.

Output from script in [Listing 2](#)

Each call to the **getHalf** function returns a value that is half the value of its incoming parameter.

As I mentioned earlier, when the function returns, the returned value replaces the call to the function. Therefore, in each case, the returned value from the call to the **getHalf** function becomes part of the argument list for the **write** method before that method is called. This causes the two calls to the **write** method in [Listing 2](#) to display the values returned from the calls to the **getHalf** function as shown in [Figure 2](#).

Figure 2 . Output from script in Listing 2.

```
Call getHalf for 10.6  
Half is: 5.3  
Call getHalf again for 12.3  
Half is: 6.15
```

Good programming design

It is a principle of good programming design that each function should perform only one task, and should perform it well. The task performed by the function named **getHalf** is to calculate and return half the value that it receives, and it does that task very well.

Arithmetic operators

The division operation in [Listing 2](#) introduced the use of arithmetic operators.

In computer programming jargon, we speak of operators and operands. Operators operate on operands.

As a real-world example, if you were to go to the hospital for knee surgery, the surgeon would be the **operator** and you would be the **operand** . The surgeon would operate on you.

Binary operators

The operators that we will use in the modules in this collection will usually be restricted to those that have two operands, a **left operand** and a **right operand** . (Operators with two operands are commonly called **binary operators** .)

In the function named **getHalf** in [Listing 2](#), the "/" character is the division operator. The left operand is **incomingParameter** and the right operand is **2**. The result is that the incoming parameter is divided by the right operand (2).

Binary arithmetic operators

The binary arithmetic operators supported by JavaScript are shown in [Figure 3](#).

Figure 3 . Binary arithmetic operators.

+ Addition:	Adds the operands
- Subtraction:	Subtracts the right operand from the left operand
* Multiplication:	Multiplies the operands
/ Division:	Divides the left operand by the right operand
% Modulus:	Returns integer remainder of dividing the left operand by the right operand

We will use these operators extensively as we work through the physics exercises in future modules.

Sequence

Of the four items listed under [Structured programming](#).earlier, the simplest one is **sequence** .

The concept of sequence in structured programming simply means that code statements can be executed in sequential order. [Listing 2](#) provides a good example of the sequential execution of statements. The code in [Listing 2](#) shows four sequential statements that begin with **document.write**.

Selection

The next item that we will discuss from the list under [Structured programming](#).is **selection** . While not as simple as **sequence** , selection is something that you probably do many times each day without even thinking about it. Therefore, once you understand it, it isn't complicated.

General syntax for selection statement

The general syntax for a selection statement (often called an **if-else** statement) is shown in [Figure 4](#).

Figure 4 . General syntax for selection statement.

```
if(conditional expression is true){  
    execute code  
}else{//optional  
    execute alternative code  
}//end selection statement
```

A selection statement performs a logical test that returns either true or false. Depending on the result, specific code is executed to control the behavior of the script.

A real-world analogy

[Figure 5](#) shows a real-world analogy of a selection statement that you might make on your day off.

Figure 5 . Real-world analogy of a selection statement.

```
if(it is not raining){  
    Play tennis  
    Go for a walk  
    Relax on the beach  
}else{//optional  
    Make popcorn  
    Watch TV  
}//end selection statement
```

In this analogy, you would check outside to confirm that it is not raining. If the condition is true (meaning that it isn't raining), you would play tennis, go for a walk, and then relax on the beach. If it is raining, (meaning that the test condition is false), you would make some popcorn and relax in front of the TV.

The else clause is optional

Note that when writing JavaScript code, the **else** clause shown in [Figure 5](#) is optional. In other words, you might choose to direct the script to take some

specific action if the condition is true, but simply transfer control to the next sequential statement in the script if the condition is false.

A selection script example

[Listing 3](#) shows a sample script containing two selection statements (commonly called **if-else** statements) in sequence.

Listing 3 . A selection script example.

```
<!-- File JavaScript03.html -->
<html><body>
<script language="JavaScript1.3">

if(3 > 2){
document.write("3 is greater than 2.,"<br/>")
}else{
document.write("3 is not greater than 2.")
}//end if

if(3 < 2){
document.write("3 is less than 2.,"<br/>")
}else{
document.write("3 is not less than 2.")
}//end if

</script>
</body></html>
```

If 3 is greater than 2...

The conditional clause in the first **if** statement in Listing 3 uses the "greater-than" relational operator ">" to determine if the literal value 3 is greater than the literal value 2, producing the first line of output shown in [Figure 6](#).

Figure 6 . Output from script in Listing 3.

```
3 is greater than 2.  
3 is not less than 2.
```

The test returns true

Since 3 is always greater than 2, the statement in the line immediately following the first test in [Listing 3](#) is executed and the code in the line following the word **else** is skipped producing the first line of output text shown in [Figure 6](#).

If 3 is less than 2...

The conditional clause in the second **if** statement in [Listing 3](#) uses the "less-than" relational operator (see [Figure 7](#)) to determine if the literal value 3 is less than the literal value 2.

The test returns false

Since 3 isn't less than 2, the statement in the line immediately following the second test in [Listing 3](#) is skipped and the statement in the line immediately following the second word **else** is executed producing the second line of output text shown in [Figure 6](#).

Relational and logical operators

I doubt that I will a frequent need to use logical operators in the modules in this collection. If I do, I will explain them at the time. However, I will use relational operators.

The relational operators that are supported by JavaScript are shown in [Figure 7](#).

Figure 7 . Relational operators.

```
> Left operand is greater than right operand
>= Left operand is greater than or equal to
right operand
< Left operand is less than right operand
<= Left operand is less than or equal to right
operand
== Left operand is equal to right operand
!= Left operand is not equal to right operand
```

As with the arithmetic operators discussed earlier, we will use these operators extensively as we work through the physics exercises in future modules.

Variables

The next item in the list under [Structured programming](#) is **repetition** . Before I can explain repetition, however, I need to explain the use of

variables.

What is a variable?

You can think of a variable as the symbolic name for a pigeonhole in memory where the script can store a value. The script can change the values stored in that pigeonhole during the execution of the script. Once a value has been stored in a variable, that value can be accessed by calling out the name of the variable.

Variable names

Variable names must conform to the naming rules for identifiers. A JavaScript identifier must start with a letter or underscore "_". Following this, you can use either letters or the symbols for the digits (0-9) in the variable name.

JavaScript is case sensitive. Therefore letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Declaring a variable

In many languages, including JavaScript, you must *declare* a variable before you can use it. However, JavaScript is very loose in this regard. There are two ways to declare a variable in JavaScript:

- By simply assigning it a value; for example, `x = 10`
- By using the keyword **var** ; for example, **var** `x = 10`

Scope

When working with variables in JavaScript and other languages as well, you must always be concerned about an issue known as scope. Among other things, scope determines which statements in a script have access to a variable.

Two kinds of variables

JavaScript recognizes two kinds of variables:

- local
- global

Local variables are variables that are declared inside a function. Global variables are variables that are declared outside a function.

Scope

Local variables may only be accessed by other code within the same function following the declaration of the variable. Hence, the scope of local variables is limited to the function or method in which it is declared.

Global variables may be accessed by any code within the script following the declaration of the variable. Hence, the scope of global variables is the entire script.

Use of the keyword `var`

Using **`var`** to declare a global variable is optional. However, you must use **`var`** to declare a variable inside a function (a local variable).

A sample script that uses variables

A sample script that uses variables to store data is shown in [Listing 4](#).

Listing 4 . A sample script that uses variables.

Listing 4 . A sample script that uses variables.

```
<!-- File JavaScript04.html -->
<html><body>
<script language="JavaScript1.3">

//Define the function named getArea()
function getArea(theRadius) {
    //declare a local variable
    var theArea

    //calculate the area
    theArea = Math.PI * theRadius * theRadius
    return theArea
} //end function getArea()
//=====//

//declare a global variable without keyword
var
area = getArea(3.2) //call the function
document.write("Area is: " + area)

</script>
</body></html>
```

The area of a circle

[Listing 4](#) begins by defining a function that will compute and return the area of a circle given the radius of the circle as an incoming parameter.

The code in the function begins by declaring a variable named **theArea** . Effectively, this declaration sets aside a pigeon hole in memory and gives it the name **theArea** . Once the variable is declared, it can be accessed by calling out its name. It can be used to store data, or it can be used to retrieve data previously stored there.

Behavior of the function named `getArea`

You may recall that the area of a circle is calculated by multiplying the mathematical constant PI by the radius squared. There is no squaring operator in JavaScript. Therefore, you can square a value by multiplying it by itself.

PI times the square of the radius

The code in the function named `getArea` in [Listing 4](#) computes the area and uses the **assignment** operator "=" to store the result in the variable named **`theArea`** . This statement represents a case where a value is being stored in a variable (assigned to the variable) for safekeeping.

Then the code in the function extracts the value stored in the variable named **`theArea`** and returns that value. After that, the function terminates and returns control to the place in the calling script from which it was originally called.

The variable named `area`

Further down the page in [Listing 4](#), the script declares a variable named **`area`** without using the keyword **`var`** . (Note, however, that the keyword **`var`** could have been used here. I prefer that approach for reasons that I won't get into here.)

The script calls the `getArea` function, passing a radius value of 3.2 as a parameter. As you learned earlier, the value returned by the function replaces the call to the function, which is then assigned to the variable named **`area`** .

Display the results

Then the script calls the **`write`** method to display some text followed by the value stored in the variable named **`area`** , producing the output shown in [Figure 8](#) in the browser window.

Figure 8 . Output from script in Listing 4.

```
Area is: 32.169908772759484
```

The string concatenation operator

The code in [Listing 4](#) exposes another operator that I will refer to as the *string concatenation operator* .

Note the argument list for the call to the **write** method in [Listing 4](#). In addition to being used to perform numeric addition, the plus operator "+" can be used to concatenate (join) two strings.

If two strings are joined by the + operator, the two strings will produce a new string that replaces the combination of the two original strings and the + operator.

If the left operand to the + operator is a string and the right operand is a numeric value (or the name of a variable containing a numeric value), the numeric value will be replaced by a string of characters that represent that numeric value and the two strings will be concatenated into a single string.

Repetition

The last item in the list under [Structured programming](#) is **repetition** , and that will be the topic of this section.

Repetition (also referred to as looping) means the act of causing something to repeat.

A repetition or loop is a set of commands that executes repeatedly until a specified condition is met.

JavaScript supports three loop statements:

- while
- for
- do-while

The while loop

The **while** loop is not only the simplest of the three, it is also the most fundamental. It can be used to satisfy any requirement for repetition in a JavaScript script. The other two exist solely for added convenience in some situations. Therefore, I will concentrate on the **while** loop, and leave the other two to be discussed in a future module, if at all.

Loop while a condition is true

A **while** loop executes the statements in its body for as long as a specified condition evaluates to true. The general syntax of a **while** loop is shown in [Figure 9](#).

Figure 9 . General syntax for a while loop.

```
while(condition is true){  
    //Execute statements in body of loop.  
}//end while statement
```

When the condition is no longer true...

When the conditional expression evaluates to false, control passes to the next statement following the **while** loop.

while loops can be nested inside of other statements, including other **while** loops.

An infinite loop

As with all loop statements, you must be careful to make certain that the conditional expression eventually evaluates to false. Otherwise, control will be trapped inside the **while** loop in what is commonly called an infinite loop.

A simple while loop

The script in [Listing 5](#) illustrates the use of a simple **while** loop.

Listing 5 . A simple while loop.

```
<!-- File JavaScript05.html -->
<html><body>
<script language="JavaScript1.3">

cnt = 4//initialize a counter variable
while(cnt >= 0){//begin while loop
    //display value of counter
    document.write("cnt = " + cnt + "<br/>")
    cnt = cnt - 1//decrement counter
}//end while loop

document.write("The End.");

</script>
</body></html>
```

A counting loop

This script initializes the value of a counter variable named **cnt** to 4. Control continues to loop (iterate) for as long as the value of the counter is greater than or equal to zero. During each iteration of the loop, the current value of **cnt** is displayed and then the value of **cnt** is reduced by a value of 1.

Repeat the test

At that point, control returns to the top of the loop where the test is repeated. This process produces the first five lines of output text shown in [Figure 10](#).

Figure 10 . Output from script in Listing 5.

```
cnt = 4
cnt = 3
cnt = 2
cnt = 1
cnt = 0
The End.
```

When the test returns false...

When the test in [Listing 5](#) returns false, meaning that **cnt** is no longer greater than or equal to zero, control exits the **while** loop and goes to the next statement following the **while** loop. This is the statement that calls the **write** method to display the last line of text in [Figure 10](#).

Programming errors

From time to time, we all make errors when writing scripts or programs. Typical errors include typing a period instead of a comma, failing to include a matching right parenthesis, etc. Usually, when you make a programming error using JavaScript, some or all of the script simply doesn't execute.

Finding and fixing the errors

The worst thing about programming errors is the need to find and fix them. The Firefox and Google Chrome browsers have easy-to-use mechanisms to help you identify the cause of the problem so that you can fix it. Internet Explorer probably has similar capability, but so far, I haven't figured out how to access it.

My recommendation is to simply open the files containing your JavaScript code in either Firefox or Google Chrome. Or, you can open the file in Internet Explorer, but if it doesn't do what you expect it to do, open it again in Google Chrome or Firefox for assistance in finding and fixing the problem.

Assistance using Google Chrome

You can open the JavaScript console in the Chrome browser by holding down the Ctrl key and the Shift key and pressing the J key. The console will open at the bottom of the Chrome browser window. You can also close the console with the same keystroke.

The format of the console is a little messy and may be difficult to navigate. However, it can be useful in locating errors if you can navigate it.

An error message in the console

If you open an html file containing a JavaScript error in the browser while the console is open, an error message will appear in the console. For example, I am looking at such an error as I type this document. It consists of a round red circle with a white x followed by the following text:

"Uncaught SyntaxError: Unexpected number"

The file name and line number

On the far right side of the same line is text that reads `junk.html:23`. That is the name of the file and the line number in that file containing the error. That text is a hyperlink. If the hyperlink is selected, another part of the console opens showing the offending line of JavaScript code.

The description is unreliable

Also, in this particular case the description of the error isn't very useful in determining the cause of the error although sometimes it may be useful. My advice is not to put too much faith in that description. The error was actually a missing relational operator in a comparison clause.

The line number is very important

Probably the most useful information is the line number that you can use to go back and examine your source code, looking for an error in that line of code.

Assistance using Firefox

You can open an error console when using the Firefox browser by holding down the Ctrl key and the Shift key and pressing the J key. The console will open in a separate window. Unlike with Chrome, repeating the keystroke won't close the error console.

An error message in the console

If you open an html file containing a JavaScript error in the browser while the error console is open, an error message will appear in the console. For example, I am looking at such an error as I type this document. It consists of a round red circle with a white x and the following text:

missing) after condition

file: --html file name and path here-- Line: 23

```
while(h 0){
```

The middle line is a hyperlink

The middle line of text that contains the file name to the left of the line number is a hyperlink. If you select the link, a window will open showing the source code with the problem line highlighted. Pressing the right arrow key will cause a blinking cursor to appear between the first and second characters in that line.

The description is unreliable

As with Chrome, in this particular case the description of the error isn't very useful in determining the cause of the error although sometimes it may be useful. My advice is not to put too much faith in that description. The error was actually a missing relational operator in a comparison clause.

The line number is very important

Probably the most useful information is the line number that you can use to go back and examine your source code, looking for an error in that line of code.

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0310 JavaScript
- File: Game0310.htm
- Published: 10/12/12
- Revised: 12/27/14

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0320 Brief Trigonometry Tutorial

GAME 2302-0320 Brief Trigonometry Tutorial

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion](#)
 - [Degrees versus radians](#)
 - [Sine, cosine, and tangent](#)
 - [The sine and arcsine of an angle](#)
 - [The cosine and arccosine of an angle](#)
 - [The tangent and arctangent of an angle](#)
 - [Dealing with quadrants](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Output for script in Listing 1.
- [Figure 2](#). A 3-4-5 triangle.
- [Figure 3](#). Output for script in Listing 2.
- [Figure 4](#). Output for script in Listing 3.
- [Figure 5](#). Interesting sine equations.
- [Figure 6](#). Interesting cosine equations.
- [Figure 7](#). Output for script in Listing 5
- [Figure 8](#). Two very important equations.
- [Figure 9](#). Interesting tangent equations.
- [Figure 10](#). Output for script in Listing 7.
- [Figure 11](#). Sinusoidal values at 90-degree increments.
- [Figure 12](#). Sinusoidal values at 45-degree increments.
- [Figure 13](#). Sinusoidal values at 22.5-degree increments.
- [Figure 14](#). Plot of cosine and sine curves.
- [Figure 15](#). Algebraic signs versus quadrants.
- [Figure 16](#). Output from the code in Listing 9.

Listings

- [Listing 1](#). Conversions between radians and degrees.
- [Listing 2](#). Arcsin of 3-4-5 triangle.
- [Listing 3](#). Finding length of the opposite side.
- [Listing 4](#). Arccosine of 3-4-5 triangle.
- [Listing 5](#). Finding the length of the adjacent side.
- [Listing 6](#). Arctan of 3-4-5 triangle.
- [Listing 7](#). Finding the length of the opposite side.
- [Listing 8](#). Sinusoidal amplitude versus angle.

- [Listing 9](#). A function to deal with quadrants.

General background information

Many of the computational requirements for an introductory physics course involve trigonometry. This module provides a brief tutorial on trigonometry fundamentals.

Sine, cosine, and tangent

There are many topics, such as identities, that are covered in an introductory trigonometry course that won't be covered in this module. Instead, this module will concentrate mainly on performing computations on right angles using the sine, cosine, and tangent of an angle.

If I find it necessary to deal with identities in a later module, I will come back and update this module accordingly.

Discussion

Degrees versus radians

The most common unit of angular measurement used by the general public is the degree. As you are probably aware, there are 360 degrees in a circle.

The most common unit of angular measurement used by scientists and engineers is the radian.

(If you would like more background on radians, go to <http://www.clarku.edu/~djoyce/trig/>.)

Conversions between radians and degrees

You may or may not be aware that one radian is equal to approximately 57.3 degrees. It is easier to remember, however, that 180 degrees is equal to π radians where π is the mathematical constant having an approximate value of 3.14159. We will use this latter relationship extensively to convert from

degrees to radians and to convert from radians to degrees while working through the exercises in these modules.

An exercise involving degrees and radians

Let's do a short exercise involving degrees and radians. Please create an html file containing the code shown in [Listing 1](#) and open it in your browser.

Listing 1 . Conversions between radians and degrees.

Listing 1 . Conversions between radians and degrees.

```
<!-- File JavaScript01.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var degrees = 90
var radians = toRadians(degrees)
document.write("degrees = " + degrees + "
               " : radians = " + radians + "
<br/>")
radians = 1
degrees = toDegrees(radians)
document.write("radians = " + radians + "
               " : degrees = " + degrees + "
<br/>")

radians = Math.PI
degrees = toDegrees(radians)
document.write("radians = " + radians + "
               " : degrees = " + degrees)

</script>
</body></html>
```

Output for script in Listing 1

When you open the file in your browser, the text shown in [Figure 1](#) should be displayed in the browser window.

Figure 1 . Output for script in Listing 1.

```
degrees = 90 : radians = 1.5707963267948965  
radians = 1 : degrees = 57.29577951308232  
radians = 3.141592653589793 : degrees = 180
```

The toRadians and toDegrees functions

Because it will frequently be necessary for us to convert between degrees and radians, I decided to write two functions that we will use to make those conversions. That will eliminate the need for us to stop and think about the conversion (and possibly get it backwards) when writing code. We will simply call the function that performs the conversion in the required direction.

The **toRadians** function expects to receive an input parameter describing an angle in degrees and returns the value for that same angle in radians.

The **toDegrees** function expects to receive an input parameter describing an angle in radians and returns the value for that same angle in degrees.

Global variables named degrees and radians

The code in [Listing 1](#) begins by declaring global variables named **degrees** and **radians** . The variable named **degrees** is initialized to 90 degrees.

The **toRadians** function is called to convert that value of degrees to radians. The returned value in radians is stored in the variable named **radians** .

Display contents of both variables

Then the **document.write** method is called to display the values contained in both variables, producing the first line of output text shown in [Figure 1](#).

Modify variable contents, convert, and display again

Following that, a value of 1 is assigned to the variable named **radians** . The **toDegrees** function is called to convert that value to degrees, and the result is stored in the variable named **degrees** .

Once again, the **document.write** method is called to display the current contents of both variables, producing the second line of output text shown in [Figure 1](#).

One more time

Finally, the mathematical constant, PI, is stored in the variable named **radians** . Then that value is converted to degrees and stored in the variable named **degrees** . The current values in both variables are displayed, producing the last line of output text shown in [Figure 1](#).

And the results were...

As you can see from [Figure 1](#),

- Ninety degrees is equal to 1.57 radians
- One radian is equal to 57.296 degrees
- 3.14 (PI) radians is equal to 180 degrees

A template

You might want to save your html file as a template for use with future exercises that require conversions between radians and degrees. This will be particularly useful when we write scripts that use JavaScript's built-in trigonometric methods. Those methods deal with angles almost exclusively

in radians while we tend to think of angles in degrees. We will use these two functions to perform conversions between degrees and radians when required.

Sine, cosine, and tangent

An exercise involving a right triangle

For the next exercise, I would like for you to use pencil and graph paper to draw a right triangle formed by points at the following coordinates:

- The origin
- $x=3, y=0$
- $x=3, y=4$

The vertices of a right triangle

Each point represents a vertex of a right triangle. If you connect the imaginary points with lines, you will have "drawn" a right triangle with its base on the horizontal axis.

The base of the triangle will have a length of three units. Another side of the triangle will have a length of four units. The hypotenuse of the triangle will have still another length.

The angle at the origin

The base and the hypotenuse will form an angle at the origin opening outward to the right. As mentioned before, the base will be on the horizontal axis. The side that connects the base and the far end of the hypotenuse will be parallel to the vertical axis.

Names for the three sides of the right triangle

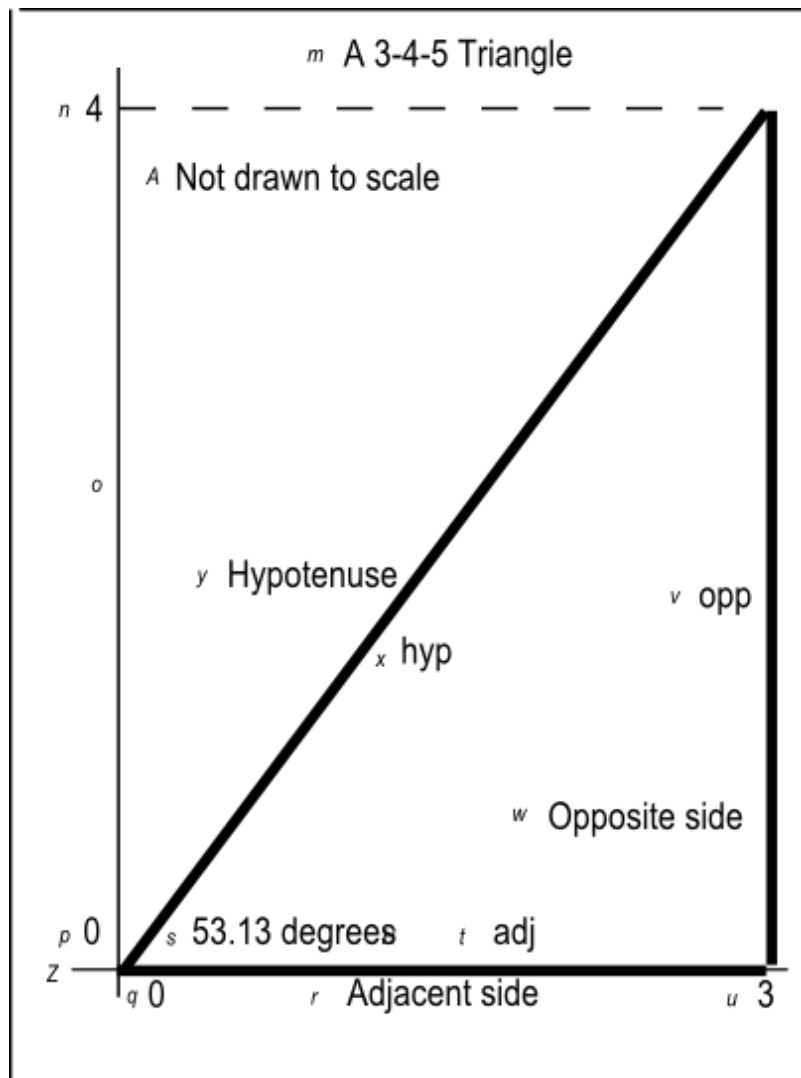
Lets establish some names for the three sides of the right triangle when drawn in this manner.

We will continue to refer to the hypotenuse as the hypotenuse and abbreviate it **hyp** . We will refer to the base as the *adjacent* side relative to the angle at the origin and abbreviate it **adj** . We will refer to the third side as the *opposite* side relative to the angle at the origin and abbreviate it **opp** .

[Figure 2](#) shows such a 3-4-5 triangle (*which may not be drawn to scale*) using the notation given above for the three sides.

(Note that this image, and many of the other images in this collection of physics modules were re-purposed from an earlier set of modules. The lower-case characters that you will see scattered throughout the images were needed there but will generally have no meaning in these modules. They should be ignored.)

Figure 2 - A 3-4-5 triangle.



The length of the hypotenuse

Now that you have your right triangle and you know the lengths of the adjacent and opposite sides, do you remember how to calculate the length of the hypotenuse?

The Pythagorean theorem

Hopefully you know that for a right triangle, the square of the hypotenuse is equal to the sum of the squares of the two other sides. Thus, the length of the hypotenuse is equal to the square root of the sum of the squares of the other two sides.

In this case we can do the arithmetic in our heads to compute the length of the hypotenuse. (I planned it that way.)

The square of the adjacent side is 9. The square of the opposite side is 16. The sum of the squares is 25, and the square root of 25 is 5. Thus, the length of the hypotenuse is 5.

A 3-4-5 triangle

You have created a rather unique triangle. You have created a right triangle in which the sides are either equal to, or proportional to the integer values 3, 4, and 5.

I chose this triangle on purpose for its simplicity. We will use it to investigate some aspects of trigonometry.

The sine and arcsine of an angle

You will often hear people talk about the sine of an angle or the cosine of an angle. Just what is the sine of an angle anyway?

Although the sine of an angle is based on very specific geometric considerations involving circles (see <http://www.clarku.edu/~djoyce/trig/>), for our purposes, the sine of an angle is simply a ratio between the lengths of two different sides of a right triangle.

A ratio of two sides

For our purposes, we will say that the sine of an angle is equal to the ratio of the opposite side and the hypotenuse. Therefore, in the case of the 3-4-5 triangle, the **sine of the angle** at the origin is equal to $4/5$ or 0.8 .

If we know the lengths of the hypotenuse and the opposite side, we can compute the sine and use it to determine the value of the angle. (We will do this later using the arcsine.)

Conversely, if we know the value of the angle but don't know the lengths of the hypotenuse and/or the opposite side, we can obtain the value of the sine of the angle using a scientific calculator (such as the Google calculator) or lookup table.

Note:

The sine of an angle -- sample computation

Enter the following into the Google search box:

$\sin(53.13010235415598 \text{ degrees})$

The following will appear immediately below the search box:

$\sin(53.13010235415598 \text{ degrees}) = 0.8$

This matches the value that we computed [above](#) as the ratio of the opposite side and the hypotenuse.

The arcsine (inverse sine) of an angle

The arcsine of an angle is the value of the angle having a given sine value. In other words, if you know the value of the sine of an unknown angle, you can use a scientific calculator or lookup table to find the value of the angle.

For example, we know that the sine of the angle at the origin on your 3-4-5 triangle is $4/5$. From that, we can determine the value of the angle. However, we probably can't do this calculation in our heads so we will use the Google calculator to compute the value of the angle.

Note:

The arcsine of an angle -- sample computation

Enter the following into the Google search box:

$\arcsin(4/5)$ in degrees

The following will appear immediately below the search box:

$\arcsin(4/5) = 53.1301024 \text{ degrees}$

This is the angle that corresponds to a ratio of the opposite side to the hypotenuse of 4/5.

We can also write a JavaScript script to perform the calculation, which we will do shortly.

Getting the angle for a known sine value

If you have access to a protractor, use it to measure and record the angle at the origin on your triangle. Then create an html file containing the code shown in [Listing 2](#) and open it in your browser.

Listing 2 . Arcsin of 3-4-5 triangle.

Listing 2 . Arcsin of 3-4-5 triangle.

```
<!-- File JavaScript02.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var opp = 4
var hyp = 5
var ratio = opp/hyp
var angRad = Math.asin(ratio)
var angDeg = toDegrees(angRad)

document.write("radians = " + angRad + "<br/>")
document.write("degrees = " + angDeg)

</script>
</body></html>
```

The output for the angle

When you open your html file in your browser, the output shown in [Figure 3](#) should appear in the browser window.

Figure 3 . Output for script in Listing 2.

```
radians = 0.9272952180016123  
degrees = 53.13010235415598
```

Did you measure the angle to be 53 degrees with your protractor. If so, congratulations. If not, you should probably take another look at it.

Define conversion functions

The code in [Listing 2](#) begins by defining the functions named **toRadians** and **toDegrees** that we developed earlier in [Listing 1](#). (In this case, we will only need the function named **toDegrees** so I could have omitted the code for the function named **toRadians** .)

Declare and initialize variables

Then the code in [Listing 2](#) declares and initializes variables to represent the lengths of the opposite side and the hypotenuse for the triangle (**opp** and **hyp**). Then it computes and saves the ratio of the two. (We learned earlier that the ratio is the value of the sine of the angle at the origin even though we don't know the value of the angle.)

The built-in Math.asin method

JavaScript has a built-in method named **Math.asin** that receives the sine value for an unknown angle and returns the value of the corresponding angle in radians. (The **Math.asin** method has the same purpose as the word arcsin in the Google calculator.)

The returned value is an angle between $-\pi/2$ and $\pi/2$ radians. (I will have more to say about this later.)

[Listing 2](#) calls the **Math.asin** method, passing the ratio (sine of the angle) as a parameter, and stores the returned value in a variable named **angRad** .

Then [Listing 2](#) calls the **toDegrees** method, passing the value of **angRad** as a parameter and stores the returned value in a variable named **angDeg** .

Finally, [Listing 2](#) calls the **document.write** method twice in success to display the angle values shown in [Figure 3](#).

Another exercise with a different viewpoint

Now let's approach things from a different viewpoint. Assume that

- you know the value of the angle in degrees,
- you know the length of the hypotenuse, and
- you need to find the length of the opposite side.

Assume also that for some reason you can't simply measure the length of the opposite side. Therefore, you must calculate it. This is a common situation in physics, so let's see if we can write a script that will perform that calculation for us.

Create an html file containing the code shown in [Listing 3](#) and open the file in your browser.

Listing 3 . Finding length of the opposite side.

Listing 3 . Finding length of the opposite side.

```
<!-- File JavaScript03.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var hyp = 5
var angDeg = 53.13
var angRad = toRadians(angDeg)
var sine = Math.sin(angRad)

var opp = hyp * sine

document.write("opposite = " + opp + "<br/>")

hyp = opp/sine

document.write("hypotenuse = " + hyp + "<br/>")

</script>
</body></html>
```

The output for the opposite side

When you open your html file in your browser, the output shown in [Figure 3](#) should appear in your browser window.

Figure 4 . Output for script in Listing 3.

```
opposite = 3.999994640742543  
hypotenuse = 5
```

Computing length of opposite side with the Google calculator

We could also compute the length of the opposite side using the Google calculator.

Note:

The length of the opposite side -- sample computation

Enter the following into the Google search box:

$5 \cdot \sin(53.1301024 \text{ degrees})$

The following will appear immediately below the search box:

$5 \cdot \sin(53.1301024 \text{ degrees}) = 4$

This is the length of the opposite side for the given angle and the given length of the hypotenuse.

Interesting equations

We learned earlier that the sine of the angle is equal to the ratio of the opposite side and the hypotenuse. We also learned that the angle is the

arcsine of that ratio.

If we know any two of those values (**angle** , **opp** , **hyp**), we can find the third (with a little algebraic manipulation) as shown in [Figure 5](#).

Figure 5 . Interesting sine equations.

```
sine(angle) = opp/hyp  
  
angle = arcsine(opp/hyp)  
opp = hyp * sine(angle)  
hyp = opp/sine(angle)
```

Getting back to Listing 3

After defining the radian/degree conversion functions, [Listing 3](#) declares and initializes variables representing the length of the hypotenuse and the angle in degrees. (Note that the angle in degrees was truncated to four significant digits, which may introduce a slight inaccuracy into the computations.)

Get and use the sine of the angle

That angle is converted to radians and passed as a parameter to the **Math.sin** method, which returns the value of the sine of the angle.

The value for the sine of the angle is then used in an algebraic equation to compute the length of the opposite side, which is displayed in [Figure 4](#). (This equation is one of the equations shown in [Figure 5](#).)

Looks very close to me

As you can see, the computed value for the opposite side shown in [Figure 4](#) is extremely close to the known value of 4 units.

Re-compute the length of the hypotenuse

After that, the value of the hypotenuse is re-computed (as though it were the unknown in the problem) using the value of the sine and the recently computed value of the opposite side. (Once again, one of the equations from [Figure 5](#) is used to perform the computation.) The output length for the hypotenuse is shown in [Figure 4](#), and it matches the known value.

Example usage of Math.asin and Math.sin methods

[Listing 2](#) and [Listing 3](#) provide examples of how to use the JavaScript **Math.asin** and **Math.sin** methods to find the angle, the opposite side, or the hypotenuse of a right triangle when the other two are known as shown by the equations in [Figure 5](#).

The cosine and arccosine of an angle

You are going to find the discussion in this section to be very similar to the discussion in the previous section on the sine and the arcsine of an angle.

Once again, although the cosine of an angle is based on very specific geometric considerations involving circles (see <http://www.clarku.edu/~djoyce/trig/>), for our purposes, the cosine of an angle is simply a ratio between the lengths of two different sides of a right triangle.

A ratio of two sides

For our purposes, we will say that the cosine of an angle is equal to the ratio of the adjacent side and the hypotenuse. Therefore, in the case of the 3-4-5 triangle, the cosine of the angle at the origin is equal to $3/5$ or 0.6.

As before, if we know the lengths of the hypotenuse and the adjacent side, we can compute the cosine and use it to determine the value of the angle.

(We will do this later.)

Conversely, if we know the value of the angle but don't know the lengths of the hypotenuse and/or the adjacent side, we can obtain the cosine value (the ratio of the adjacent side and the hypotenuse) using a scientific calculator or lookup table and use it for other purposes later.

Note:

The cosine of an angle -- sample computation

Enter the following into the Google search box:

$\cos(53.13010235415598 \text{ degrees})$

The following will appear immediately below the search box:

$\cos(53.13010235415598 \text{ degrees}) = 0.6$

This matches the ratio of the adjacent side to the hypotenuse for a 3-4-5 triangle.

The arccosine (inverse cosine) of an angle

The arccosine of an angle is the value of the angle having a given cosine value. In other words, if you know the value of the cosine of an unknown angle, you can use a scientific calculator or lookup table to find the value of the angle.

Getting the angle for a known cosine value

For example, we know that the cosine of the angle at the origin of the 3-4-5 triangle is 0.6. From that, we can determine the value of the angle using either the Google calculator or JavaScript.

Note:

The arccosine of an angle -- sample computation

Enter the following into the Google search box:

$\arccos(3/5)$ in degrees

The following will appear immediately below the search box:

$\arccos(3/5) = 53.1301024$ degrees

This is the angle that corresponds to a ratio of the adjacent side to the hypotenuse of 3/5.

As you should expect, the computed angle is the same as before. We didn't change the angle, we simply computed it using a different approach.

Getting the angle using JavaScript

Please create an html file containing the code shown in [Listing 4](#) and open it in your browser.

Listing 4 . Arccosine of 3-4-5 triangle.

Listing 4 . Arccosine of 3-4-5 triangle.

```
<!-- File JavaScript04.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var adj = 3
var hyp = 5
var ratio = adj/hyp
var angRad = Math.acos(ratio)
var angDeg = toDegrees(angRad)

document.write("radians = " + angRad + "<br/>")
document.write("degrees = " + angDeg)

</script>
</body></html>
```

Similar to a previous script

If you examine the code in [Listing 4](#) carefully, you will see that it is very similar to the code in [Listing 2](#) with a couple of exceptions:

- The variable **opp** having a value of 4 was replaced by the variable **adj** having a value of 3.

- The call to the **Math.asin** method was replaced by a call to the **Math.acos** method.

The output

When you load your html file into your browser, it should produce the output shown earlier in [Figure 3](#). In other words, we know that the angle at the origin didn't change. What changed was the manner in which we computed the value of that angle.

Different approaches to the same solution

In [Listing 2](#), we used the length of the hypotenuse and the length of the opposite side, along with the arcsine method to compute the angle.

In [Listing 4](#), we used the length of the hypotenuse and the length of the adjacent side, along with the arccosine method to compute the angle.

Which approach should you use?

As would be expected, since the angle didn't change, both approaches produced the same result. Deciding which approach to use often depends on the values that are available to use in the computation.

Sometimes you only have the lengths of the hypotenuse and the opposite side available, in which case you could use the arcsine. Sometimes you only have the lengths of the hypotenuse and the adjacent side available, in which case you could use the arccosine. Sometimes you have the lengths of both the opposite side and the adjacent side in addition to the length of the hypotenuse, in which case you can use either approach.

Both approaches use the length of the hypotenuse

It is important to note however that both of these approaches require you to have the length of the hypotenuse. Later in this module we will discuss the tangent and arctangent for an angle, which allows us to work with the opposite side and the adjacent side devoid of the length of the hypotenuse. (Of course, if you have the lengths of the opposite side and the adjacent side,

you can always find the length of the hypotenuse using the Pythagorean theorem.)

Interesting cosine equations

The equations in [Figure 6](#) are similar to equations in [Figure 5](#). The difference is that the equations in [Figure 5](#) are based on the use of the sine of the angle and the opposite side whereas the equations in [Figure 6](#) are based on the use of the cosine of the angle and the adjacent side.

As you can see in [Figure 6](#), if you know any two of the values for **angle**, **adj**, and **hyp**, you can find the other value. This is illustrated in the script shown in [Listing 5](#), which produces the output shown in [Figure 7](#).

Figure 6 . Interesting cosine equations.

```
cosine(angle) = adj/hyp  
  
angle = arccosine(adj/hyp)  
adj = hyp * cosine(angle)  
hyp = adj/cosine(angle)
```

Finding the length of the adjacent side

The code in [Listing 5](#) is very similar to the code in [Listing 2](#). The main difference is that [Listing 2](#) is based on the use of the sine of the angle and the length of the opposite side whereas [Listing 5](#) is based on the use of the cosine of the angle and the length of the adjacent side.

Listing 5 . Finding the length of the adjacent side.

```
<!-- File JavaScript05.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var hyp = 5
var angDeg = 53.13
var angRad = toRadians(angDeg)
var cosine = Math.cos(angRad)

var adj = hyp * cosine

document.write("adjacent = " + adj + "<br/>")

hyp = adj/cosine

document.write("hypotenuse = " + hyp + "<br/>")

</script>
</body></html>
```

No further explanation needed

Because of the similarity of [Listing 5](#) and [Listing 2](#), no further explanation of the code in [Listing 5](#) should be needed. As you can see from [Figure 7](#), the output values match the known lengths for the hypotenuse and the adjacent side for the 3-4-5 triangle.

Figure 7 . Output for script in Listing 5.

```
adjacent = 3.0000071456633126  
hypotenuse = 5
```

Computing length of adjacent side with the Google calculator

We could also compute the length of the adjacent side using the Google calculator.

Note:

The length of the adjacent side -- sample computation

Enter the following into the Google search box:

$5 \cdot \cos(53.1301024 \text{ degrees})$

The following will appear immediately below the search box:

$5 \cdot \cos(53.1301024 \text{ degrees}) = 3$

This is the length of the adjacent side for the given angle and the given length of the hypotenuse.

Two very important equations

From an introductory physics viewpoint, two of the most important and perhaps most frequently used equations from [Figure 5](#) and [Figure 6](#) are shown in [Figure 8](#).

Figure 8 . Two very important equations.

$$\begin{aligned}\text{opp} &= \text{hyp} * \text{sine}(\text{angle}) \\ \text{adj} &= \text{hyp} * \text{cosine}(\text{angle})\end{aligned}$$

These two equations are so important that it might be worth your while to memorize them. Of course, you will occasionally need most of the equations in [Figure 5](#) and [Figure 6](#), so you should try to remember them, or at least know where to find them when you need them.

Vectors

As you will see later in the module that deals with vectors, you are often presented with something that resembles the hypotenuse of a right triangle whose adjacent side is on the horizontal axis and whose opposite side is parallel to the vertical axis.

The thing that looks like the hypotenuse of a right triangle is called a **vector**. It has a length and it has a direction. Typically, the direction is stated as the angle between the vector and the horizontal axis. Thus, the direction is analogous to the angle at the origin in your triangle.

Horizontal and vertical components

For reasons that I won't explain until we get to that module, you will often need to compute the horizontal and vertical components of the vector. The

horizontal component is essentially the adjacent side of our current right triangle. Thus, the value of the horizontal component can be computed using the second equation in [Figure 8](#).

The vertical component is essentially the opposite side of our current right triangle, and its value can be computed using the first equation in [Figure 8](#).

The tangent and arctangent of an angle

Once again, although the tangent of an angle is based on very specific geometric considerations involving circles (see <http://www.clarku.edu/~djoyce/trig/>), for our purposes, the tangent of an angle is simply a ratio between the lengths of two different sides of a right triangle.

A ratio of two sides

For our purposes, we will say that the tangent of an angle is equal to the ratio of the opposite side and the adjacent side. Therefore, in the case of the 3-4-5 triangle, the **tangent of the angle at the origin is equal to** $4/3$ or 1.333.

Not limited to 1.0

Note that the absolute value for the sine and the cosine of an angle is limited to a maximum value of 1.0. However, the tangent of an angle is not so limited. In fact, the tangent of 45 degrees is 1.0 and the tangent of 90 degrees is infinity. This results from the length of the adjacent side, which is the denominator in the ratio, going to zero at 90 degrees.

Dividing by zero in a script is usually not a good thing. This is a pitfall that you must watch out for when working with tangents. I will provide code later that shows you how deal with this issue.

Computing the tangent

If we know the lengths of the opposite side and the adjacent side, we can compute the tangent and use it for other purposes later without having to know the value of the angle.

Conversely, if we know the value of the angle but don't know the lengths of the adjacent side and/or the opposite side, we can obtain the tangent value using a scientific calculator or lookup table and use it for other purposes later.

Note:

The tangent of an angle -- sample computation

Enter the following into the Google search box:

$\tan(53.13010235415598 \text{ degrees})$

The following will appear immediately below the search box:

$\tan(53.13010235415598 \text{ degrees}) = 1.33333333$

This agrees with the ratio that we computed [earlier](#).

The arctangent (inverse tangent) of an angle

The arctangent of an angle is the value of the angle having a given tangent value. (For example, as mentioned above, the arctangent of infinity is 90 degrees and the arctangent of 1.0 is 45 degrees.) In other words, if you know the value of the tangent of an unknown angle, you can use a scientific calculator or lookup table to find the value of the angle.

For example, we know that the tangent of the angle at the origin on your 3-4-5 triangle is 1.333. From that, we can determine the value of the angle.

Note:

The arctangent of an angle -- sample computation

Enter the following into the Google search box:

$\arctan(4/3)$ in degrees

The following will appear immediately below the search box:

```
arctan(4/3) = 53.1301024 degrees
```

We can also write a JavaScript script to perform the calculation.

Getting the angle for a known tangent value using JavaScript

Please create an html file containing the code shown in [Listing 6](#) and open it in your browser.

Listing 6 . Arctan of 3-4-5 triangle.

Listing 6 . Arctan of 3-4-5 triangle.

```
<!-- File JavaScript06.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var opp = 4
var adj = 3
var ratio = opp/adj
var angRad = Math.atan(ratio)
var angDeg = toDegrees(angRad)

document.write("radians = " + angRad + "<br/>")
document.write("degrees = " + angDeg)

</script>
</body></html>
```

The output from the script

Once again, when you open this file in your browser, the output shown in [Figure 3](#) should appear in your browser window.

The code in [Listing 6](#) is very similar to the code in [Listing 2](#). They both describe the same right triangle, so the output should be the same in both

cases.

The code in [Listing 2](#) uses the opposite side and the hypotenuse along with the arcsine to compute the angle. The code in [Listing 6](#) uses the opposite side and the adjacent side along with the arctangent to compute the angle. Otherwise, no further explanation should be required.

Interesting tangent equations

In the spirit of [Figure 5](#) and [Figure 6](#), [Figure 9](#) provides some interesting equations that deal with the angle, the opposite side, and the adjacent side. Given any two, you can find the third using either the tangent or arctangent.

Figure 9 . Interesting tangent equations.

```
tangent(angle) = opp/adj  
  
angle = arctangent(opp/adj)  
opp = tangent(angle) * adj  
adj = opp/tangent(angle)
```

An exercise involving the tangent

Please copy the code from [Listing 7](#) into an html file and open it in your browser.

Listing 7 . Finding the length of the opposite side.

```
<!-- File JavaScript07.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
    return radians*180/Math.PI
}//end function toDegrees
//=====//

var adj = 3
var angDeg = 53.13
var angRad = toRadians(angDeg)
var tangent = Math.tan(angRad)

var opp = adj * tangent

document.write("opposite = " + opp + "<br/>")

adj = opp/tangent

document.write("adjacent = " + adj + "<br/>")

</script>
</body></html>
```

When you open your html file in your browser, the output shown in [Figure 10](#) should appear in your browser window. We can see that the values in [Figure 10](#) are correct for our 3-4-5 triangle.

Figure 10 . Output for script in Listing 7.

```
opposite = 3.9999851132269173  
adjacent = 3
```

Very similar code

The code in [Listing 7](#) is very similar to the code in [Listing 3](#) and [Listing 5](#). The essential differences are that

- [Listing 3](#) uses the sine along with the opposite side and the hypotenuse.
- [Listing 5](#) uses the cosine along with the adjacent side and the hypotenuse.
- [Listing 7](#) uses the tangent along with the opposite side and the adjacent side.

You should be able to work through those differences without further explanation from me.

The cotangent of an angle

There is also something called the cotangent of an angle, which is simply the ratio of the adjacent side to the opposite side. If you know how to work with the tangent, you don't ordinarily need to use the cotangent, so I won't discuss it further.

Computing length of opposite side with the Google calculator

We could also compute the length of the opposite side using the Google calculator.

Note:

The length of the opposite side -- sample computation

Enter the following into the Google search box:

```
3*tan(53.1301024 degrees)
```

The following will appear immediately below the search box:

```
3 * tan(53.1301024 degrees) = 4.00000001
```

Dealing with different quadrants

Up to this point, we have dealt exclusively with angles in the range of 0 to 90 degrees (the first quadrant). As long as you stay in the first quadrant, things are relatively straightforward.

As you are probably aware, however, angles can range anywhere from 0 to 360 degrees (or more). Once you begin working with angles that are greater than 90 degrees, things become a little less straightforward.

Sinusoidal amplitude versus angle

Please copy the code from [Listing 8](#) into an html file and open the file in your browser.

Listing 8 . Sinusoidal amplitude versus angle.

```
<!-- File JavaScript08.html -->
<html><body>
<script language="JavaScript1.3">

function toRadians(degrees){
    return degrees*Math.PI/180
}//end function toRadians
//=====//

function toDegrees(radians){
```

Listing 8 . Sinusoidal amplitude versus angle.

```
    return radians*180/Math.PI
} //end function toDegrees
//=====//

var angInc = 90
var angStart = -360
var ang = angStart
var angEnd = 360
var sine
var cosine

while(ang <= angEnd){
    //Compute sine and cosine of angle
    sine = Math.sin(toRadians(ang))
    cosine = Math.cos(toRadians(ang))

    //Reduce the number of digits in the output
    sine = (Math.round(100*sine))/100
    cosine = (Math.round(100*cosine))/100

    //Display the results
    document.write("Angle: " + ang +
                   " Sine: " + sine +
                   " Cosine: " + cosine +
                   "<\/br>")

    //Increase the angle for next iteration
    ang = ang + angInc
} //end while loop

<\/script>
<\/body><\/html>
```

Output from the script

When you open your html file in your browser, the output shown in [Figure 11](#) should appear in your browser window.

Figure 11 . Sinusoidal values at 90-degree increments.

```
Angle: -360 Sine: 0 Cosine: 1
Angle: -270 Sine: 1 Cosine: 0
Angle: -180 Sine: 0 Cosine: -1
Angle: -90 Sine: -1 Cosine: 0
Angle: 0 Sine: 0 Cosine: 1
Angle: 90 Sine: 1 Cosine: 0
Angle: 180 Sine: 0 Cosine: -1
Angle: 270 Sine: -1 Cosine: 0
Angle: 360 Sine: 0 Cosine: 1
```

[Figure 11](#) contains the data for two different curves. One is a sine curve and the other is a cosine curve.

Plot the points

You should be able to plot these data values as two separate curves on your graph paper.

Remember, the angle values from -360 degrees (-2π radians) to +360 degrees ($+2\pi$ radians) are horizontal coordinates while the corresponding values for the sine and cosine are vertical coordinates.

Saw tooth curves

Once you have plotted the points, you should be able to discern two curves, each of which is a saw tooth.

The two curves have exactly the same shape, but one is shifted horizontally relative to the other. For example, the sine curve has a value of zero at an angle of zero (the origin) and it is asymmetric about the vertical axis.

The cosine curve, on the other hand has a value of 1 at an angle of zero and it is symmetric about the vertical axis.

Periodic curves

These are periodic curves. For example, the shape of the sine curve between -360 and 0 is the same as the shape of the sine curve between 0 and +360. Each of those ranges represents one *cycle* of the periodic curve.

We only computed the values from -360 to +360. However, if we had computed the values from -3600 to + 3600, the overall shape of the curve would not differ from what we have here. The shape of each cycle of the curve would be identical to the shape of the cycle to the left and the cycle to the right.

Not really a saw tooth

The sine and cosine curves don't really have a saw tooth shape. That is an artifact of the fact that we didn't compute enough points to reliably describe the shape of the curves. Let's improve on that.

Modify the script

Modify the code in your script to initialize the value of the variable named **angInc** to 45 degrees instead of 90 degrees and then load the revised version into your browser. This will cause the script to fill in data points between the points that we already have producing the output shown in [Figure 16](#).

Figure 12 . Sinusoidal values at 45-degree increments.

Figure 12 . Sinusoidal values at 45-degree increments.

```
Angle: -360 Sine: 0 Cosine: 1
Angle: -315 Sine: 0.71 Cosine: 0.71
Angle: -270 Sine: 1 Cosine: 0
Angle: -225 Sine: 0.71 Cosine: -0.71
Angle: -180 Sine: 0 Cosine: -1
Angle: -135 Sine: -0.71 Cosine: -0.71
Angle: -90 Sine: -1 Cosine: 0
Angle: -45 Sine: -0.71 Cosine: 0.71
Angle: 0 Sine: 0 Cosine: 1
Angle: 45 Sine: 0.71 Cosine: 0.71
Angle: 90 Sine: 1 Cosine: 0
Angle: 135 Sine: 0.71 Cosine: -0.71
Angle: 180 Sine: 0 Cosine: -1
Angle: 225 Sine: -0.71 Cosine: -0.71
Angle: 270 Sine: -1 Cosine: 0
Angle: 315 Sine: -0.71 Cosine: 0.71
Angle: 360 Sine: 0 Cosine: 1
```

Plot the new points

Every other line of text in [Figure 12](#) should contain sine and cosine values for angles that are half way between the points that you already have plotted.

Plot the new points and connect all of the points in each curve.

Same shape but shifted horizontally

The two curves still have the same shape, although shifted horizontally relative to one another and they are still periodic. However, they no longer have a saw tooth shape. They tend to be a little more rounded near the peaks

and they are beginning to provide a better representation of the actual shapes of the sine and cosine curves.

Let's do it again

Change the value of the variable named **angInc** from 45 degrees to 22.5 degrees and load the new version of the html file into your browser. Now the output should look like [Figure 13](#).

Figure 13 . Sinusoidal values at 22.5-degree increments.

```
Angle: -360 Sine: 0 Cosine: 1
Angle: -337.5 Sine: 0.38 Cosine: 0.92
Angle: -315 Sine: 0.71 Cosine: 0.71
Angle: -292.5 Sine: 0.92 Cosine: 0.38
Angle: -270 Sine: 1 Cosine: 0
Angle: -247.5 Sine: 0.92 Cosine: -0.38
Angle: -225 Sine: 0.71 Cosine: -0.71
Angle: -202.5 Sine: 0.38 Cosine: -0.92
Angle: -180 Sine: 0 Cosine: -1
Angle: -157.5 Sine: -0.38 Cosine: -0.92
Angle: -135 Sine: -0.71 Cosine: -0.71
Angle: -112.5 Sine: -0.92 Cosine: -0.38
Angle: -90 Sine: -1 Cosine: 0
Angle: -67.5 Sine: -0.92 Cosine: 0.38
Angle: -45 Sine: -0.71 Cosine: 0.71
Angle: -22.5 Sine: -0.38 Cosine: 0.92
Angle: 0 Sine: 0 Cosine: 1
Angle: 22.5 Sine: 0.38 Cosine: 0.92
Angle: 45 Sine: 0.71 Cosine: 0.71
Angle: 67.5 Sine: 0.92 Cosine: 0.38
Angle: 90 Sine: 1 Cosine: 0
```

Figure 13 . Sinusoidal values at 22.5-degree increments.

```
Angle: 112.5 Sine: 0.92 Cosine: -0.38
Angle: 135 Sine: 0.71 Cosine: -0.71
Angle: 157.5 Sine: 0.38 Cosine: -0.92
Angle: 180 Sine: 0 Cosine: -1
Angle: 202.5 Sine: -0.38 Cosine: -0.92
Angle: 225 Sine: -0.71 Cosine: -0.71
Angle: 247.5 Sine: -0.92 Cosine: -0.38
Angle: 270 Sine: -1 Cosine: 0
Angle: 292.5 Sine: -0.92 Cosine: 0.38
Angle: 315 Sine: -0.71 Cosine: 0.71
Angle: 337.5 Sine: -0.38 Cosine: 0.92
Angle: 360 Sine: 0 Cosine: 1
```

A lot of data points

Once again, every other line of text in [Figure 13](#) contains new sine and cosine values for angles that you don't have plotted yet.

Plotting all of these point is going to require a lot of effort. Before you do that, let's think about it.

Two full cycles

You should have been able to discern by now that your plots for the sine and cosine graphs each contain two full cycles. An important thing about periodic functions is that once you know the shape of the curve for any one cycle, you know the shape of the curve for every cycle from minus infinity to infinity. The shape of every cycle is exactly the same as the shape of every other cycle.

Saving patience and effort

If you are running out of patience, you might consider updating your plots for only one cycle.

You should be able to discern that your curves no longer have a saw tooth shape. Each time we have run the script, we have sampled the amplitude values of each curve at twice as many points as before. Therefore, the curves should be taking on a smoother rounded shape that is better representation of the actual shape of the curves.

Continue the process

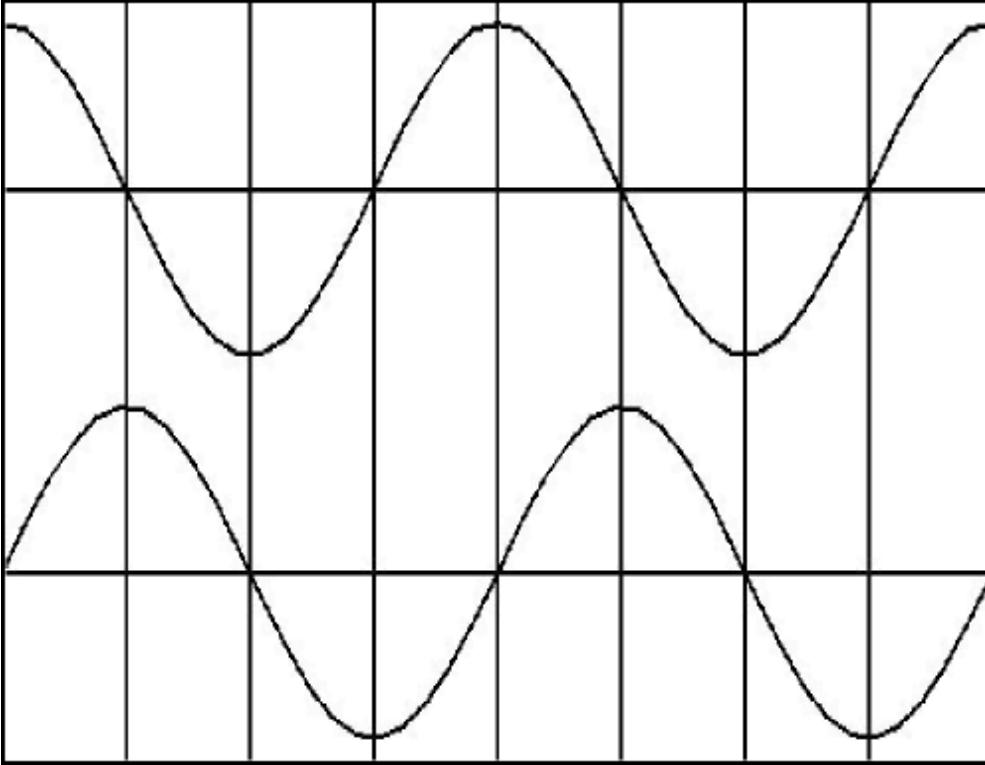
You can continue this process of improving the curves for as long as you have the space and patience to do so. Just divide the value of the variable named **angInc** by a factor of two and rerun the script. That will produce twice as many data points that are only half as far apart on the horizontal axis.

If you choose to do so, you can plot only the new points in one-half of a cycle to get an idea of the shape. By now you should have discerned that each half of a cycle has the same shape, only one half is above the horizontal axis and the other half is a mirror image below the axis.

Plot of cosine and sine curves

[Figure 14](#) shows a cosine curve plotted above a sine curve very similar to the curves that you have plotted on your graph paper.

Figure 14 - Plot of cosine and sine curves.



Grid lines

The image in [Figure 14](#) contains 7 vertical grid lines. The vertical grid line in the center represents an angle of zero degrees. The space between each grid line on either side of the center represents an angle of 90 degrees or $\pi/2$ radians.

There are two horizontal grid lines. One is one-fourth of the way down from the top. The other is one-fourth of the way up from the bottom.

The curves

A cosine curve is plotted with vertical values relative to the top grid line. It extends from -360 degrees on the left to +360 degrees on the right.

A sine curve is plotted with vertical values relative to the bottom grid line. It also extends from -360 degrees on the left to +360 degrees on the right.

Return values for the `Math.asin`, `Math.acos`, and `Math.atan` methods

I told you earlier that the **Math.asin** method returns a value between $-\pi/2$ and $\pi/2$. However, I didn't tell you that the **Math.acos** method returns a value between 0 and π , or that the **Math.atan** method returns a value between $-\pi/2$ and $\pi/2$. You now have enough information to understand why this is true.

Smooth curves

If you examine the two curves in [Figure 14](#) and the data in [Figure 13](#), you can surmise that the sine and cosine functions are smooth curves whose values range between -1 and +1 inclusive. For every possible value between -1 and +1, there is an angle in the range $-\pi/2$ and $\pi/2$ whose sine value matches that value. There is also an angle in the range 0 and π whose cosine value matches that value.

(Although you haven't plotted the curve for the tangent, a similar situation holds there also, but the maximum value is not limited to 1.0.)

An infinite number of angles

Therefore, given a specific numeric value between -1 and +1, there are an infinite number of angles whose sine and cosine values match that numeric value and the method has no way of distinguishing between them.

Therefore, the **Math.asin** method returns the matching angle that is closest to zero and the **Math.acos** method returns the matching positive angle that is closest to zero.

What can we learn from this?

One important thing that we can learn is there is no difference between the sine or cosine of an angle and the sine or cosine of a different angle that differs from the original angle by 360 degrees. Thus, the **Math.asin** and **Math.acos** methods cannot be used to distinguish between angles that differ by 360 degrees. (As you learned above, the situation involving the **Math.asin** and **Math.acos** methods is even more stringent than that.)

One-quarter cycle contains all of the information

Another thing that we can learn is that once you know the shape of the cosine curve from 0 degrees to 90 degrees, you have enough information to construct the entire cosine curve and the entire sine curve across any range of angles. Every possible value or the negative of every possible value that can occur in a sine or cosine curve occurs in the cosine curve between 0 degrees and 90 degrees. Furthermore, the order of those values is also well defined.

Think about these relationships

You should think about these kinds of relationships. As I mentioned earlier, as long as we are working with angles between 0 and 90 degrees, everything is relatively straightforward. However, once we start working with angles between 90 degrees and 360 degrees (or greater), things become a little less straightforward.

If you have a good picture in your mind of the shape of the two curves between -360 degrees and +360 degrees, you may be able to avoid errors once you start working on physics problems that involve angles outside the range of 0 to 90 degrees.

Quadrants

We often think of a two-dimensional space with horizontal and vertical axes and the origin at the center in quadrants. Each quadrant is bounded by half the horizontal axis and half the vertical axis.

It is common practice to number the quadrants in counter-clockwise order with the upper-right quadrant being quadrant 1, the upper-left quadrant being quadrant 2, the bottom-left quadrant being quadrant 3, and the bottom-right quadrant being quadrant 4.

Angles fall in quadrants

If you measure the angle between the positive horizontal axis and a line segment that emanates from the origin, quadrant 1 contains angles between 0 and $\pi/2$, quadrant 2 contains the angles between $\pi/2$ and π , quadrant 3 contains the angles between π and $3\pi/2$, and quadrant 4 contains the

angles between $3\pi/2$ and 2π (or zero). (Note that I didn't attempt to reconcile the inclusion of each axis in the two quadrants on either side of the axis.)

Algebraic signs versus quadrant number

It is sometimes useful to consider how the algebraic sign of the sine, cosine, and tangent values varies among the four quadrants. [Figure 15](#) contains a table that shows the sign of the sine, cosine, and tangent values for each of the four quadrants

Figure 15 . Algebraic signs versus quadrants.

	1	2	3	4
sine	+	+	-	-
cosine	+	-	-	+
tangent	+	-	+	-

Working with arctangents is more difficult than arcsine or arccosine

Working with arctangent is somewhat more difficult than working with arcsine or arccosine, if for no other reason than the possibility of dividing by zero when working with the arctangent.

[Listing 9](#) shows a JavaScript function named `getAngle` that deals with this issue.

Listing 9 . Listing 9: A function to deal with quadrants.


```

<!------- File JavaScript09.html -----
----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/>");

//The purpose of this function is to receive the
adjacent
// and opposite side values for a right triangle
and to
// return the angle in degrees in the correct
quadrant.
function getAngle(adjacent,opposite){
    if((adjacent == 0) && (opposite == 0)){
        //Angle is indeterminate. Just return zero.
        return 0;
    }else if((adjacent == 0) && (opposite > 0)){
        //Avoid divide by zero denominator.
        return 90;
    }else if((adjacent == 0) && (opposite < 0)){
        //Avoid divide by zero denominator.
        return -90;
    }else if((adjacent < 0) && (opposite >= 0)){
        //Correct to second quadrant
        return Math.atan(opposite/adjacent)*180/Math.PI
+ 180;
    }else if((adjacent < 0) && (opposite <= 0)){
        //Correct to third quadrant
        return Math.atan(opposite/adjacent)*180/Math.PI
+ 180;
    }else{
        //First and fourth quadrants. No correction
required.
        return
Math.atan(opposite/adjacent)*180/Math.PI;
    }//end else

```

```
}//end function getAngle
```

```
//Modify these values and run for different cases.
```

```
var adj = 3;
```

```
var opp = 4;
```

```
document.write("adj = " + adj.toFixed(2) +  
" opp = " + opp.toFixed(2) + " units<br/>");
```

```
document.write("angle = " +  
getAngle(adj,opp).toFixed(2)  
+ " units<br/>");
```

```
var adj = -3;
```

```
var opp = 4;
```

```
document.write("adj = " + adj.toFixed(2) +  
" opp = " + opp.toFixed(2) + " units<br/>");
```

```
document.write("angle = " +  
getAngle(adj,opp).toFixed(2)  
+ " units<br/>");
```

```
var adj = -3;
```

```
var opp = -4;
```

```
document.write("adj = " + adj.toFixed(2) +  
" opp = " + opp.toFixed(2) + " units<br/>");
```

```
document.write("angle = " +  
getAngle(adj,opp).toFixed(2)  
+ " units<br/>");
```

```
var adj = 3;
```

```
var opp = -4;
```

```
document.write("adj = " + adj.toFixed(2) +  
" opp = " + opp.toFixed(2) + " units<br/>");  
  
document.write("angle = " +  
getAngle(adj,opp).toFixed(2)  
+ " units<br/>");  
  
</script>  
</body></html>
```

The code in [Listing 9](#) begins by defining a function named `getAngle` that accepts the signed values of the adjacent side and the opposite side of the right triangle and returns the angle that the hypotenuse makes with the positive horizontal axis.

Then the code in [Listing 9](#) tests the result for four different triangles situated in each of the four quadrants.

[Figure 16](#) shows the output produced by this script.

Figure 16 . Output from the code in Listing 9.

```
Start Script  
adj = 3.00 opp = 4.00 units  
angle = 53.13 units  
adj = -3.00 opp = 4.00 units  
angle = 126.87 units  
adj = -3.00 opp = -4.00 units  
angle = 233.13 units  
adj = 3.00 opp = -4.00 units  
angle = -53.13 units
```

This is an issue that will become important when we reach the module that deals with vectors in all four quadrants.

Structure of the script

The script shown in [Listing 9](#) begins by defining a function named **getAngle**. The purpose of this function is to return an angle in degrees in the correct quadrant based on the lengths of the adjacent and opposite sides of an enclosing right triangle. As mentioned above, the returned angle is the angle that the hypotenuse makes with the positive horizontal axis.

An indeterminate result

The **getAngle** function calls the **Math.atan** method to compute the angle whose tangent is the ratio of the opposite side to the adjacent side of a right triangle.

If the lengths of both the opposite and adjacent sides are zero, the ratio opposite/adjacent is indeterminate and the value of the angle cannot be computed. In fact there is no angle corresponding to the ratio 0/0. However, the function must either return the value of an angle, or must return some sort of flag indicating that computation of the angle is not possible.

In this case, the function simply returns the value zero for the angle.

Avoiding division by zero

If the length of adjacent side is zero and the length of opposite side is not zero, the ratio opposite/adjacent is infinite. Therefore, the value of the angle cannot be computed. However, in this case, the angle is known to be 90 degrees (for opposite greater than zero) or 270 degrees (-90 degrees, for opposite less than zero). The **getAngle** function traps both of those cases and returns the correct angle in each case.

Correcting for the quadrant

The **Math.atan** method receives one parameter and it is either a positive or negative value. If the value is positive, the method returns an angle between 0 and 90 degrees. If the value is negative, the method returns an angle

between 0 and -90 degrees. Thus, the angles returned by the **Math.atan** method always lie in the first or fourth quadrants.

(Actually, as I mentioned earlier, +90 degrees and -90 degrees are not possible because the tangent of +90 degrees or -90 degrees is an infinitely large positive or negative value. However, the method can handle angles that are very close to +90 or -90 degrees.)

A negative opposite/adjacent ratio

If the opposite/adjacent ratio is negative, this doesn't necessarily mean that the angle lies in the fourth quadrant. That negative ratio could result from a positive value for opposite and a negative value for adjacent. In that case, the angle would lie in the second quadrant between 90 degrees and 180 degrees.

The **getAngle** function tests the signs of the values for opposite and adjacent. If the signs indicate that the angle lies in the second quadrant, the value returned from the **Math.atan** method is corrected to place the angle in the second quadrant. The corrected angle is returned by the **getAngle** function.

A positive opposite/adjacent ratio

Similarly, if the opposite/adjacent ratio is positive, this doesn't necessarily mean that the angle lies in the first quadrant. That positive ratio could result from a negative opposite value and a negative adjacent value. In that case, the angle would lie in the third quadrant between 180 degrees and 270 degrees.

Again, the **getAngle** function tests the signs of the values for opposite and adjacent. If both values are negative, the value returned from the **Math.atan** method is corrected to place the angle in the third quadrant.

No corrections required...

Finally, if no corrections are required for the quadrant, the **getAngle** function returns the value returned by the **Math.atan** method. Note however, that in all cases, the **Math.atan** method returns the angle in radians. That value is

converted to degrees by the `getAngle` function and the returned value is in degrees.

Positive and negative angles

As you can see from the results of the test shown in [Figure 16](#), angles in the first, second, and third quadrants are returned as positive angles in degrees. However, angles in the fourth quadrant are returned as negative angles in degrees.

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of `.html`. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0320 Brief Trigonometry Tutorial
- File: Game0320.htm
- Published: 10/12/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it

possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0330 Scale Factors, Ratios, and Proportions

This module provides a brief tutorial on scale factors, ratios, and proportions.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion](#)
 - [Scale factors](#)
 - [Ratios](#)
 - [Proportions](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

The module provides a brief tutorial on scale factors, ratios, and proportions.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output for Listing #1.
- [Figure 2](#). Screen output for Listing #2.
- [Figure 3](#). Screen output for Listing #3.

Listings

- [Listing 1](#). Exercise on scale factors
- [Listing 2](#). Circumference is proportional to radius.
- [Listing 3](#). Area is proportional to radius squared.

General background information

Mathematical expressions are used in physics to describe relationships that are difficult to express in words. The expressions use algebraic symbols to represent quantities that consist of numbers and units.

Measurements are important

Conclusions that are drawn in physics and other sciences ranging from chemistry to the social sciences are often based on measurements such as length, width, weight, salinity, population, density, etc.

Each number in an equation often represents the results of a measurement, which is made in terms of a standard. The units indicate which standard was used to make the measurements.

Knowledge of units is critical

A number that is used to indicate the result of a measurement is of little value unless we know the units in which the measurement was made. For example, it isn't very useful to know that the length of an object is 125 unless we know whether the units are meters, centimeters, millimeters, or miles.

The Google calculator

Although I won't go into detail in this module, I will tell you that the Google calculator is very good at helping you to keep track of units. For example, if you enter the following expression in the Google search box,

3 ft + 1 yd + 36 inches

The following result will be displayed immediately below the search box:

$(3 \text{ ft}) + (1 \text{ yd}) + (36 \text{ inches}) = 2.7432 \text{ meters}$

Discussion

We often express the relationship between two items using a scale factor.

Scale factors

For example, we might say that that a colt doubled its weight in one year. This means that the colt's weight after one year was equal to the colt's weight at birth multiplied by a factor of two. The factor is the number by which a quantity is multiplied or divided when it is changed from one value to another.

Ratios

The factor is the ratio of the new value to the old value. Regarding the colt mentioned above, we might write:

$$\text{NewWeight} / \text{BirthWeight} = 2$$

where the slash character "/" indicates division.

Percentage increases

It is also common for us to talk about something increasing or decreasing by a given percentage value. For example, we might say that a colt increased its weight by 50-percent in one year. This is equivalent to saying:

$$\text{NewWeight} = \text{BirthWeight} * (1 + 50/100)$$

This assumes that when evaluating a mathematical expression:

- The asterisk character " * " indicates multiplication.
- Computations begin inside of the inner-most pair of matching parentheses and work outward from there.
- Multiplication and division are performed before addition and subtraction are performed.

This is typical behavior for computer programs and spreadsheets, but not necessarily for hand calculators.

Percentage decreases

In my dreams, I might say that I went on a diet and my weight decreased by 25-percent in one year. This would be equivalent to saying:

$$\text{NewWeight} = \text{OldWeight} * (1 - 25/100)$$

Exercise on scale factors

Write a script that has the following behavior. Given a chalk line that is 100 inches long, draw other chalk lines that are:

- A. Twice the length of the original line.
- B. Twenty-five percent of the length of the original line.
- C. Twenty-five percent greater than the length of the original line.
- D. Twenty-five percent less than the length of the original line.

My version of the script is shown in [Listing 1](#).

Listing 1 . Exercise on scale factors

```
<!-- File JavaScript01.html -->
<html><body>
<script language="JavaScript1.3">

//Do the computations
var origLine = 100
var lineA = 2 * origLine
var lineB = (25/100) * origLine
var lineC = (1 + 25/100) * origLine
var lineD = (1 - 25/100) * origLine

//Display the results
document.write("Original line = "
               + origLine + "<br/>")
document.write("A = " + lineA + "<br/>")
document.write("B = " + lineB + "<br/>")
document.write("C = " + lineC + "<br/>")
document.write("D = " + lineD + "<br/>")

</script>
</body></html>
```

Screen output

When you copy the code from [Listing 1](#) into an html file and open the file in your web browser, the output text shown in [Figure 1](#) should appear in

your browser window.

Figure 1 . Screen output for Listing #1.

```
Original line = 100  
A = 200  
B = 25  
C = 125  
D = 75
```

Note that although they sound similar, specifications B and D [above](#) don't mean the same thing.

Proportions

We talk about increasing or changing a value by some factor because we can often simplify a problem by thinking in terms of proportions.

Note:

A symbol for proportionality

Physics textbooks often use a character that doesn't appear on a QWERTY keyboard to indicate "is proportional to."

I will use a "\$" character for that purpose because:

- It does appear on a QWERTY keyboard.
- It isn't typically used in mathematical expressions unless American currency is involved.

- It is not a JavaScript operator.

For example, I will write $A \propto B$ to indicate that A is proportional to B.

When we say that A is proportional to B, or

$A \propto B$

we mean that if B increases by some factor, then A must increase by the same factor.

Circumference of a circle

Let's illustrate what we mean with a couple of examples. For the first example, we will consider the circumference of a circle. Hopefully, you know that the circumference of a circle is given by the expression:

$$C = 2 * \text{PI} * r$$

where:

- C is the circumference of the circle
- PI is the mathematical constant 3.14159...
- r is the radius of the circle

From this expression, we can conclude that

$C \propto r$

If we modify the radius...

If we double the radius, the circumference will also double. If we reduce the radius by 25-percent, the circumference will also be reduced by 25-percent. This is illustrated by the script in [Listing 2](#).

Listing 2 . Circumference is proportional to radius.

```
<!-- File JavaScript02.html -->
<html><body>
<script language="JavaScript1.3">

var r = 10
var C = 2 * Math.PI * r
document.write("r =" + r +
               ", C = " + C + "<br/>")

//Multiply r by 2. Then display r and C
r = r * 2
C = 2 * Math.PI * r
document.write("r =" + r +
               ", C = " + C + "<br/>")

//Reduce r by 25%, Then display r and C
r = r * (1 - 25/100)
C = 2 * Math.PI * r
document.write("r =" + r +
               ", C = " + C + "<br/>")

</script>
</body></html>
```

Output from the script

When you open the script shown in [Listing 2](#) in your browser, the text shown in [Figure 2](#) should appear in your browser window.

Figure 2 . Screen output for Listing #2.

```
r =10, C = 62.83185307179586  
r =20, C = 125.66370614359172  
r =15, C = 94.24777960769379
```

[Figure 2](#) shows the value of the circumference for three different values for the radius. You should be able to confirm that the combination of the three lines of output text satisfy the proportionality rules stated [earlier](#).

For example, you can confirm these results by entering the following three expressions in the Google search box and recording the results that appear immediately below the search box:

$2*\pi*10$

$2*\pi*20$

$2*\pi*15$

Area of a circle

Before I can discuss the area of a circle, I need to define a symbol that we can use to indicate exponentiation.

Note:

A symbol for exponentiation

Physics textbooks typically use a superscript character to indicate that a value is raised to a power, such as the radius of a circle squared.

When I need to indicate that a value is raised to a power, I will use the "^" character, such as in the following text that indicates radius squared, or radius raised to the second power.

radius^2

In those cases where the exponent is a fraction, or is negative, I will surround it with parentheses, such as in $\text{radius}^{(1/2)}$, and $\text{distance}^{(-2)}$

The first term indicates the square root of the radius. The second term indicates that distance is being raised to the -2 power.

I chose to use the "^" character to indicate exponentiation because it is used as the exponentiation operator in some programming languages, such as BASIC. The "^" character is also recognized by the Google calculator as an exponentiation operator.

Unfortunately, there is no exponentiation operator in JavaScript, so we will need a different approach to raise a value to a power in our JavaScript scripts. As you will see later, we will use the built-in **Math.pow** method for that purpose.

An expression for the area of a circle

From your earlier coursework, you should know that the area of a circle is given by

$$A = \text{PI} * r^2$$

where

- A is the area.
- PI is the mathematical constant 3.14159...
- r is the radius of the circle.

Proportional to the square of the radius

From this, we can conclude that the area of a circle is not proportional to the radius. Instead, it is proportional to the square of the radius as in

$$A \propto r^2$$

If you change the radius...

If you change the value of the radius, the area changes in proportion to the square of the radius. If the radius doubles, the area increases by four. If the radius is decreased by 25-percent, the area decreases by more than 25-percent. This is illustrated by the script in [Listing 3](#).

Listing 3 . Area is proportional to radius squared.

Listing 3 . Area is proportional to radius squared.

```
<!-- File JavaScript03.html -->
<html><body>
<script language="JavaScript1.3">

var r = 10
var A = Math.PI * Math.pow(r,2)
document.write("r =" + r +
               ", A = " + A + "<br/>")

//Multiply r by 2. Then display r and C
r = r * 2
var A = Math.PI * Math.pow(r,2)
document.write("r =" + r +
               ", A = " + A + "<br/>")

//Reduce r by 25%, Then display r and C
r = r * (1 - 25/100)
var A = Math.PI * Math.pow(r,2)
document.write("r =" + r +
               ", A = " + A + "<br/>")

//Compute and the display the cube root
// of a number.
var X = Math.pow(8,1/3)
document.write("Cube root of 8 = " + X)

</script>
</body></html>
```

The JavaScript Math.pow method

[Listing 3](#) calls a built-in JavaScript method that I have not used before: **Math.pow** . This method is called to raise a value to a power. It requires

two parameters. The first parameter is the value that is to be raised to a power and the second parameter is the power to which the value is to be raised.

The method returns the value raised to the power.

Fractional exponents

Although this topic is not directly related to the discussion on proportionality, as long as I am introducing the method named **Math.pow**, I will point out that it is legal for the exponent to be a fraction. The last little bit of code in [Listing 3](#) raises the value 8 to the 1/3 power. This actually computes the cube root of the value 8. As you should be able to confirm in your head, the cube root of 8 is 2, because two raised to the third power is 8.

Output from the script

When you open the script shown in [Listing 3](#) in your browser, the text shown in [Figure 3](#) should appear in your browser window.

Figure 3 . Screen output for Listing #3.

```
r =10, A = 314.1592653589793
r =20, A = 1256.6370614359173
r =15, A = 706.8583470577034
Cube root of 8 = 2
```

An examination of the first three lines of text in [Figure 3](#) should confirm that they satisfy the proportionality rules for the square of the radius described [earlier](#).

The last line of text in [Figure 3](#) confirms that the **Math.pow** method can be used to compute roots by specifying fractional exponents as the second parameter.

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of .html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0330 Scale Factors, Ratios, and Proportions
- File: Game0330.htm
- Revised: 01/25/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0340 Scientific Notation and Significant Figures

The purpose of this module is to explain the use of scientific notation and significant figures.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Accuracy and precision](#)
 - [Scientific notation](#)
 - [Significant figures](#)
- [Discussion and sample code](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

The purpose of this module is to explain the use of scientific notation and significant figures.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Examples of significant figures.
- [Figure 2](#). Screen output from Listing #1.
- [Figure 3](#). Screen output from Listing #2.
- [Figure 4](#). Behavior of the toPrecision method.
- [Figure 5](#). Screen output from Listing #3.

Listings

- [Listing 1](#). An exercise involving addition.
- [Listing 2](#). An exercise involving multiplication.
- [Listing 3](#). An exercise involving combined operations.

General background information

This section will contain a discussion of accuracy, precision, scientific notation, and significant figures.

Accuracy and precision

Let's begin with a brief discussion of accuracy and precision. These two terms are often used interchangeably in everyday conversation, but they have very different meanings in the world of science and engineering.

Accuracy

In science and engineering, the accuracy of a measurement system is the degree of closeness of measurements of a quantity to its actual (true) value.

Precision

The precision of a measurement system (also called reproducibility or repeatability) is the degree to which repeated measurements under unchanged conditions show the same result.

Four possibilities

A measurement system can be:

- Both accurate and precise.
- Accurate but not precise.
- Precise but not accurate.
- Neither accurate nor precise.

A hypothetical experiment

Consider an experiment where a firearm is clamped into a fixture, very carefully aimed at a bulls eye on a downrange target, and fired six times. (Although you may never have seen or touched a firearm, you probably have a pretty good idea of how they behave.)

If the six holes produced by the bullets in the target fall in a tight cluster in the bulls eye, the system can be considered to be both accurate and precise.

If all of the holes fall in the general area of the bulls eye but the cluster is not very tight, the system can be considered to be accurate but not precise.

If all of the holes fall in a tight cluster but the cluster is some distance from the bulls eye, the system can be considered to be precise but not accurate.

If the holes are scattered across a wide area of the target, the system can be considered to be neither accurate nor precise.

Another use of the word precision

Another use of the word precision, which will be important in this module, is based on the concept that the precision of a measurement describes the units that you use to measure something.

How tall are you?

For example, if you tell someone that you are about five feet tall, that wouldn't be very precise. If you told someone that you are 62 inches tall, that would be more precise. If you told someone that you are 62.3 inches tall, that would be even more precise, and if you told someone that you are 62.37 inches tall, that would be very precise for a measurement of that nature.

The smaller the unit...

The smaller the unit you use to measure with, the more precise the measurement can be. For example, assume that you measure someone's height with a measuring stick that is longer than the person is tall. Assume also that the measuring stick is graduated only in feet. In that case, the best that you could hope for would be to get the measurement correct to the nearest foot and perhaps estimate a second digit to the nearest tenth of a foot.

One-inch graduations

On the other hand, if the measuring stick is graduated in inches and you are careful, you should be able to get the measurement correct to the nearest inch and perhaps estimate another digit to the nearest tenth of an inch. The second measurement using the one-inch graduations would be more precise than the first using the one-foot graduations.

Diminishing returns

If the measuring stick were graduated in tenth-inch units, however, that may or may not lead to a more precise measurement. That would be approaching the point of diminishing returns where your inability to take full advantage of the more precise graduations and the inability of the subject to always stand with the same degree of rigidity might come into play.

Scientific notation

According to [Wikipedia](#), *scientific notation is a way of writing numbers that accommodates values too large or small to be conveniently written in standard decimal notation* . The notation has a number of useful properties and is commonly used by scientists and engineers. A variation of scientific notation is also used internally by computers.

Scientific notation format

Numbers in scientific notation are written using the following format:

$$x * 10^y$$

which can be read as the value **x** multiplied by ten raised to the **y** power where **y** is an integer and **x** is any real number. (Constraints are placed on the value of **x** when using the **normalized** form of scientific notation which I will explain below.)

The values for **x** and **y** can be either positive or negative.

The term referred to as **x** is often called the **significand** or the **mantissa** .

The computer display version of scientific notation

Because of difficulties involved in displaying superscripts in the output of computer programs, a typical display of a number in scientific notation by a computer program might look something like the following example:

-3.141592653589793e+1

where

- either numeric value can be positive, negative, or zero,
- the number of digits in the numeric value to the left of the **e** (the mantissa) may range from a few to many, and
- the **e** may be either upper-case or lower-case depending on the computer and the program.

A power of ten is understood

In this format, it is understood that the number consists of the value to the left of the **e** (the mantissa) multiplied by ten raised to a power given by the value to the right of the **e** (the exponent).

For example, in JavaScript exponential format, the value `-10*Math.PI` is displayed as

`-3.141592653589793e+1`

The value `Math.PI/10` is displayed as

`3.141592653589793e-1`

The value `Math.PI` is displayed as

`3.141592653589793e+0`

The value `0` is displayed as

`0e+0`

The normalized form of scientific notation

Using general scientific notation, the number `-65700` could be written in several different ways including the following:

- $-6.57 * 10^4$
- $-65.7 * 10^3$
- $-657 * 10^2$

In normalized scientific notation, the exponent is chosen such that the absolute value of the mantissa is at least one but less than ten. For example, `-65700` is written:

$-6.57 * 10^4$

In normalized notation the exponent is negative for a number with absolute value between 0 and 1. For example, the value 0.00657 would be written:

$$6.57 * 10^{(-3)}$$

The 10 and the exponent are usually omitted when the exponent is 0.

Significant figures

According to [Wikipedia](#), The significant figures of a number are those digits that carry meaning contributing to its precision. This includes all digits except:

- Leading zeros where they serve merely as placeholders to indicate the scale of the number (.00356 for example).
- Spurious digits introduced, for example, by calculations carried out to greater accuracy than that of the original data, or measurements reported to a greater precision than the equipment supports.

A popular physics textbook provides a more complete set of rules for identifying the significant figures in a number:

1. Nonzero digits are always significant.
2. Final or ending zeros written to the right of the decimal point are significant.
3. Zeros written to the right of the decimal point for the purpose of spacing the decimal point are not significant.
4. Zeros written to the left of the decimal point may be significant, or they may only be there to space the decimal point. For example, 200 cm could have one, two, or three significant figures; it's not clear whether the distance was measured to the nearest 1 cm, to the nearest 10 cm, or to the nearest 100 cm. On the other hand, 200.0 cm has four significant figures (see rule 5). Rewriting the number in scientific notation is one way to remove the ambiguity.
5. Zeros written between significant figures are significant.

Ambiguity of the last digit in scientific notation

Again, according to [Wikipedia](#), it is customary in scientific measurements to record all the significant digits from the measurements, and to guess one additional digit if there is any information at all available to the observer to make a guess. The resulting number is considered more valuable than it would be without that extra digit, and it is considered a significant digit because it contains some information leading to greater precision in measurements and in aggregations of measurements (adding them or multiplying them together).

Examples of significant digits

Referring back to the physics textbook mentioned earlier, [Figure 1](#) shows:

- Four different numbers
- The number of significant figures in each number.
- The default JavaScript exponential representation of each number.

Figure 1 . Examples of significant figures.

1.	409.8	4	4.098e+2
2.	0.058700	5	5.87e-2
3.	9500	ambiguous	9.5e+3
4.	950.0 * 10^1	4	9.5e+3

Note that the default JavaScript exponential representation fails to display the significant trailing zeros for the numbers on row 2 and row 5. I will show you some ways that you may be able to deal with this issue later but you may not find them to be very straightforward.

Discussion and sample code

Beyond knowing about scientific notation and significant figures from a formatting viewpoint, you need to know how to perform arithmetic while satisfying the rules for scientific notation and significant figures.

Performing arithmetic involves **three main rules** :

1. For addition and subtraction, the final result should be rounded so as to have the same number of decimal places as the number that was included in the sum that has the smallest number of decimal places. In accordance with the discussion early in this module, this is the least precise number.
2. For multiplication and division, the final result should be rounded to have the same number of significant figures as the number that was included in the product with the smallest number of significant figures.
3. The two rules listed above should not be applied to intermediate calculations. For intermediate calculations, keep as many digits as practical. Round to the correct number of significant figures or the correct number of decimal places in the final result.

An exercise involving addition

Please copy the JavaScript code shown in [Listing 1](#) into an html file and open the file in your browser.

Listing 1 . An exercise involving addition.

Listing 1 . An exercise involving addition.

```
<!-- File JavaScript01.html -->
<html><body>
<script language="JavaScript1.3">

//Compute and display the sum of three
// numbers
var a = 169.01
var b = 0.00356
var c = 385.293
var sum = a + b + c
document.write("sum = " + sum + "<br/>")

//Round the sum to the correct number
// of digits to the right of the decimal
// point.
var round = sum.toFixed(2)
document.write("round = " + round + "<br/>")

//Display a final line as a hedge against
// unidentified coding errors.
document.write("The End")

</script>
</body></html>
```

Screen output

When you open the html file in your browser, the text shown in [Figure 2](#) should appear in your browser.

Figure 2 . Screen output from Listing #1.

```
sum = 554.30656  
round = 554.31  
The End
```

The code in [Listing 1](#) begins by declaring three variables named **a** , **b** , and **c** , adding them together, and displaying the sum in the JavaScript default format in the browser window.

Too many decimal digits

As you can see from the first line in [Figure 2](#) , the result is computed and displayed with eight decimal digits, five of which are to the right of the decimal point. We know, however, from [rule #1](#) , that we should present the result rounded to a precision of two digits to the right of the decimal point in order to match the least precise of the numbers included in the sum. In this case, the value stored in the variable named **a** is the least precise.

Correct the problem

The code in [Listing 1](#) calls a method named **toFixed** on the value stored in the variable **sum** passing a value of 2 as a parameter to the method. This method returns the value from **sum** rounded to two decimal digits. The returned value is stored in the variable named **round** . Then the script displays that value as the second line of text in [Figure 2](#) .

The output text that reads "The End"

There is a downside to using JavaScript (as opposed to other programming languages such as Java). By default, if there is a coding error in your script, there is no indication of the error in the output in the main browser window. Instead, the browser simply refuses to display some or all of the output that you are expecting to see.

Put a marker at the end

Writing the script in such a way that a known line of text, such as "The End" will appear following all of the other output won't solve coding errors. However, if it doesn't appear, you will know that there is a coding error and some or all of the output text may be missing.

JavaScript and error consoles

I explained how you can open a JavaScript console in the Google Chrome browser or an error console in the Firefox browser in an earlier module. While the diagnostic information provided in those consoles is limited, it will usually indicate the line number in the source code where the programming error was detected. Knowing the line number will help you examine the code and fix the error.

An exercise involving multiplication

Please copy the code shown in [Listing 2](#) into an html file and open it in your browser.

Listing 2 . An exercise involving multiplication.

Listing 2 . An exercise involving multiplication.

```
<!-- File JavaScript02.html -->
<html><body>
<script language="JavaScript1.3">

//Compute and display the product of three
// numbers, each having a different number
// of significant figures.
var a = 169.01
var b = 0.00356
var c = 386.253
var product = a * b * c
document.write("product = " + product + "
<br/>")

//Round the product to the correct number
// of significant figures
var rounded = product.toPrecision(5)
document.write("rounded = " + rounded + "
<br/>")

//Display a final line as a hedge against
// unidentified coding errors.
document.write("The End")

</script>
</body></html>
```

The screen output

When you open your html file in your browser, the text shown in [Figure 3](#) should appear in your browser window.

Figure 3 . Screen output from Listing #2.

```
product = 232.39900552679998  
rounded = 232.40  
The End
```

The code in [Listing 2](#) begins by declaring three variables named **a** , **b** , and **c** , multiplying them together, and displaying the product in the browser window. Each of the factors in the product have a different number of significant figures, with the factor of value 169.01 having the least number (5) of significant figures. We know from [rule #2](#) , therefore, that we need to present the result rounded to five significant figures.

The toPrecision method

[Listing 2](#) calls a method named **toPrecision** on the variable named **product** , passing the desired number of significant figures (5) as a parameter. The method rounds the value stored in **product** to the desired number of digits and returns the result, which is stored in the variable named **rounded** . Then the contents of the variable named **rounded** are displayed, producing the second line of text in [Figure 3](#) .

What about other parameter values

Note that the method named **toPrecision** knows nothing about significant figures. It was up to me to figure out the desired number of significant figures in advance and to pass that value as a parameter to the method.

Although this has nothing to do with significant figures, it may be instructive to examine the behavior of the method named **toPrecision** for several different parameter values.

[Figure 4](#) shows the result of replacing the parameter value of 5 in the call to the **toPrecision** method with the values in the first column of [Figure 4](#) and displaying the value returned by the method.

Figure 4 . Behavior of the toPrecision method.

```
1  rounded = 2e+2
2  rounded = 2.3e+2
3  rounded = 232
4  rounded = 232.4
5  rounded = 232.40
6  rounded = 232.399
7  rounded = 232.3990
10 rounded = 232.3990055
15 rounded = 232.399005526800
20 rounded = 232.39900552679998214
```

And the point is...

The point to this is to emphasize that the method named **toPrecision** is not a method that knows how to compute and display the required number of significant figures. Instead, according to the JavaScript documentation:

"The toPrecision() method formats a number to a specified length. A decimal point and nulls are added (if needed), to create the specified length."

It is up to you, the author of the script, to determine what that length should be and to provide that information as a parameter to the **toPrecision** method.

Combined operations

This is where things become a little hazy. I have been unable to find definitive information as to how to treat the precision and the number of significant figures when doing computations that combine addition and/or subtraction with multiplication and/or division.

Two contradictory procedures

I have found two procedures documented on the web that seem to be somewhat contradictory. Both sources seem to say that you should perform the addition and/or subtraction first and that you should apply [rule #1](#) to the results. However, they differ with regard to how stringently you apply that rule before moving on to the multiplication and/or division.

The more stringent procedure

One source seems to suggest that you should round the results of the addition and/or subtraction according to [rule #1](#) and replace the addition or subtraction expression in your overall expression with the rounded result. Using that approach, you simply create one the factors that will be used later in the multiplication and/or division. That factor has a well-defined number of significant figures.

Then you proceed with the multiplication and/or division and adjust the number of significant figures in the final result according to [rule #2](#).

The less stringent procedure

The other source seems to suggest that you mentally round the results of the addition and/or subtraction according to [rule #1](#) and make a note of the number of significant figures that would result if you were to actually round the result. However, you should not actually round the result at that point in time. In other words, you should use the raw result of the addition and/or subtraction as a factor in the upcoming multiplication and/or division knowing that you may be carrying excess precision according to [rule #1](#).

Then you proceed with the multiplication and/or division and adjust the number of significant figures in the final result according to [rule #2](#). However, when you adjust the number of significant figures, you should include the number of significant figures from your note in the decision process. If that is the smallest number of significant figures of all the factors, you should use it as the number of significant figures for the final result.

An exercise involving combined operations

Evaluate the following expression and display the final result with the correct number of significant figures.

$$(169.01 + 3294.6372) * (0.00365 - 29.333)$$

Please copy the code from [Listing 3](#) into an html file and open it in your browser.

Listing 3 . An exercise involving combined operations.

```
<!-- File JavaScript03.html -->
<html><body>
<script language="JavaScript1.3">

//Compute, fix the number of decimal places,
// and display the sum of two numbers.
var a1 = 169.01
var a2 = 3294.6372
var aSum = (a1 + a2).toFixed(2)
document.write("aSum = " + aSum + "<br/>")

//Compute, fix the number of decimal places,
// and display the difference between two
// other numbers.
var b1 = 0.00356
var b2 = 29.333
var bDiff = (b1 - b2).toFixed(3)
document.write("bDiff = " + bDiff + "<br/>")

//Compute and display the product of the
```

Listing 3 . An exercise involving combined operations.

```
// sum and the difference.  
var product = aSum * bDiff  
document.write("product = " + product + "  
<br/>")  
  
//Round the product to the correct number  
// of significant figures based on the least  
// number of significant figures in the  
// factors.  
var final = product.toPrecision(5)  
document.write("final = " + final + "<br/>")  
  
//Display a final line as a hedge against  
// unidentified coding errors.  
document.write("The End")  
  
</script>  
</body></html>
```

When you open your html file in your browser, the text shown in [Figure 5](#) should appear in the browser window.

Figure 5 . Screen output from Listing #3.

Figure 5 . Screen output from Listing #3.

```
aSum = 3463.65  
bDiff = -29.329  
product = -101585.39085000001  
final = -1.0159e+5  
The End
```

The more stringent procedure

The code in [Listing 3](#) implements the [more stringent procedure](#), not because it is necessarily the correct one. Rather, it is simpler to implement in a script.

Do addition and subtraction first

[Listing 3](#) begins by adding two numbers, adjusting the precision to the least precise of the two numbers, and saving the result in the variable named **aSum** .

Then [Listing 3](#) subtracts one number from another number, adjusts the precision to the least precise of the two numbers, and saves the result in the variable named **bDiff** .

Display to get information on significant figures

Both results are displayed immediately after they are obtained. This is necessary for me to know which one has the least number of significant figures. I need to know that to be able to properly adjust the number of significant figures in the final product.

In other words, it was necessary for me to write and execute the addition/subtraction portion of the script in order to get the information required to write the remainder of the script.

Do the multiplication

Then [Listing 3](#) multiplies the sum and difference values and displays the result in the default format with far too many significant figures as shown by the third line of text in [Figure 5](#).

Finally [Listing 3](#) adjusts the number of significant figures in the product based on the number of significant figures in **bDiff** and displays the final result with five significant figures in normalized scientific (exponential) notation.

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of .html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0340 Scientific Notation and Significant Figures
- File: Game0340.htm
- Published: 10/12/12
- Revised: 01/25/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0350 Units and Dimensional Analysis

The purpose of this module is to explain units and dimensional analysis.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Formatting mathematical expressions](#)
 - [What do we mean by units?](#)
 - [SI units](#)
 - [Prefixes for powers of ten](#)
 - [Dimensional analysis](#)
- [Discussion and sample code](#)
 - [Manual exercise on converting units](#)
 - [Using JavaScript to convert units](#)
 - [More substantive JavaScript examples](#)
 - [Free fall exercise](#)
 - [A plotting exercise](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

The purpose of this module is to explain units and dimensional analysis.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Single-line format.
- [Figure 2](#). SI base units.
- [Figure 3](#). Examples of SI derived units.
- [Figure 4](#). A sampling of SI prefixes.
- [Figure 5](#). Screen output for Listing #1.
- [Figure 6](#). Screen output for Listing #2.
- [Figure 7](#). Screen output for Listing #3.
- [Figure 8](#). The trajectory of the falling rock.

Listings

- [Listing 1](#). Convert from paces to miles.
- [Listing 2](#). Free fall exercise.
- [Listing 3](#). A plotting exercise.

General background information

As a young engineering student many years ago, I was told that when evaluating mathematical expressions, I should always embed the units in the expression, manipulate them algebraically, and confirm that the units that survive until the end of the evaluation are correct relative to the problem specifications.

Three good reasons

There are at least three good reasons for following this procedure:

- It shows what the units of the result are. A common mistake is to get the correct numerical result of a calculation but to write it with the wrong units.
- It shows where unit conversions must be performed. If units that should have canceled do not cancel, we need to go back and perform the necessary conversions. For example, when a speed in miles per hour is being calculated and the result comes out with units of miles per second, we should convert seconds to hours.
- It helps to locate mistakes. If a velocity (meters per second) is being calculated and the units come out as seconds per meter, we know to look for an error.

Formatting mathematical expressions

Multiple-line format

We might typically write a fraction as a horizontal line with the numerator above the line and the denominator below the line using superscripts to indicate exponents. The product of two fractions would typically be written in the same format with a multiplication indicator joining the two fractions.

Single-line format

[Figure 1](#) shows multiplication and division of fractions in a single-line format similar to how we write expressions in computer programs.

Figure 1 . Single-line format.

$$\begin{aligned} & ((3*x^2)/(4*x) * (6/(4*x) = (18*x^2)/(16*x^2) \\ & = 18/16 \end{aligned}$$

Expressions in this format may be more difficult for people to work with than the multiple line format described above. With enough time and practice, I might become almost as proficient in evaluating expressions as shown in [Figure 1](#) as in evaluating expressions in a multiple-line format, but I'm afraid that I would need a lot of time and a lot of practice to achieve that proficiency.

We must work with what we are given

However, we must work with what we are given so the mathematical expressions in this module (and most of the other modules as well) will mostly be presented in a single-line format. The good news is that the single-line format can usually be easily translated into a format that is suitable for evaluation using JavaScript and other programming languages.

What do we mean by units?

Let me try to answer this important question with an example.

If you were to walk from your home to your school and count your steps, you might report that the distance from your home to your school is 2112 steps or paces. If you are a male student, this would probably be approximately one mile, using assumptions that I will explain later. If you are a female, it would probably be somewhat less than a mile because your pace would probably be a little shorter than your male friend's pace.

A pace is a unit of measurement

In this case, the pace would be a unit of measurement. However, it would not be a *standard unit* because the length of a pace for one student is often different from the length of a pace for a different student. A male student, for example, typically has a longer pace length than a female student.

Standard units

Standard units are units whose value has been set by some official agency that has the authority to set standards. One such standard unit is the mile, which as you may know is equal to 5280 feet (more commonly pronounced feet).

A foot (not a feet) is also a standard unit, which you may also know is equal to 12 inches. An inch is another standard unit.

SI units

The discrepancy between the pronunciation of the unit **foot** and its plural **feet** is an example of the need for more consistency in the use of units. Many physics books use a system of units called **SI units**. SI is an abbreviation for a French name, which I am unable to pronounce.

I won't attempt to explain much about SI units. You can probably Google SI units and find hundreds of web pages that explain the system in varying levels of detail.

Tables of SI units

Most of those references will probably also provide tables for the units. I will also attempt to provide tables in [Figure 2](#) and [Figure 3](#).

Don't ignore the details

Note, however, that you should not ignore the online SI-unit tables altogether. They are likely to contain important information that I won't reproduce here, such as the fact that the meter is a unit of length and its

value is *the length of the path travelled by light in a vacuum during a time interval of 1/299792458 of a second* .

Base units and derived units

When reading about SI units, you will find that they are often divided into base units and derived units. I will put the base units in [Figure 2](#) and some sample derived units in [Figure 3](#).

Figure 2 . SI base units.

Base Quantity	Name	Symbol
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd

Note that the list of derived units in [Figure 3](#) is only a sampling of different units that can be derived from the base units.

The exponentiation indicator

As is the case throughout these modules, the character "^" that you see used extensively in [Figure 3](#) indicates that the character following the ^ is an exponent. Note also that when the exponent is negative, it is enclosed along with its minus sign in parentheses for clarity.

Figure 3 . Examples of SI derived units.

area	square meter	m^2
volume	cubic meter	m^3
speed, velocity	meter per second	m/s
acceleration	meter per second squared	m/s^2
wave number	reciprocal meter	m^{-1}
mass density	kilogram per cubic meter	kg/m^3
specific volume	cubic meter per kilogram	m^3/kg
current density	ampere per square meter	A/m^2

Prefixes for powers of ten

As an alternative to explicitly writing powers of ten, SI uses prefixes for units to indicate power of ten factors. [Figure 4](#) shows some of the powers of ten and the SI prefixes used for them.

Figure 4 . A sampling of SI prefixes.

Figure 4 . A sampling of SI prefixes.

Prefix-(abbreviation)	Power of Ten
peta-(P)	10^{15}
tera-(T)	10^{12}
giga-(G)	10^9
mega-(M)	10^6
kilo-(k)	10^3
deci-(d)	10^{-1}
centi-(c)	10^{-2}
milli-(m)	10^{-3}
micro-(Greek letter mu)	10^{-6}
nano-(n)	10^{-9}
pico-(p)	10^{-12}
femto-(f)	10^{-15}

Dimensional analysis

As typically used in physics, the word *dimensions* means basic types of units such as time, length, and mass (see [Figure 2](#)). *(This is a different meaning than the common meaning of the word dimensions in computer programming -- such as an array with three dimensions.)*

There are many different units that are used to express length, for example, such as foot, mile, meter, inch, etc.

Because they all have dimensions of length, you can convert from one to another. For example one mile is equal to 5280 feet.

Cannot convert between units of different dimensions

When evaluating mathematical expressions, we can add, subtract, or equate quantities only if they have the same dimensions (although they may not necessarily be given in the same units). For example, it is possible to add 3 meters to 2 inches (after converting units), but it is not possible to add 3 meters to 2 kilograms.

To analyze dimensions, you should treat them as algebraic quantities using the same procedures that we will use in the upcoming exercise on manually converting units.

Discussion and sample code

Let's work through an exercise manually to determine the distance from your home to your school in miles given that we know the distance in paces. (I will convert this to a JavaScript script later in the module.)

As we discussed earlier, you have already determined that the distance from your home to your school is 2112 paces. We have also recognized that the length of one of your paces is likely to be different from the length of someone else's paces.

Manual exercise on converting units

This is a somewhat long and tedious explanation, but unless you already have quite a lot of experience in this area, you should probably bear with me as I walk you through it.

Convert your pace length to standard units

First we must convert the length of your pace to standard units, which we can then convert to miles.

A calibration exercise

Assume that you go into an empty parking lot, mark your starting position, walk 100 paces, and then mark your ending position on the pavement.

Assume that you then use a measuring tape to measure the distance from the starting position to the ending position and you find that distance to be 3000 inches.

From that, we could conclude that for your pace length,

$$100 \text{ paces} = 3000 \text{ inches}$$

Thus, we have calibrated your pace length in terms of the standard unit inch.

The plural versus the singular

Note that the use of the plural such as **inches** and the singular such as **inch** has no impact on the results of our computations. we can switch back and forth from the plural to the singular at will as long as it makes sense to do so.

A conversion factor

If we divide both sides of the above [equation](#) by 100 paces, we get

$$1 = 3000 \text{ inches} / 100 \text{ paces} = 30 \text{ inches/pace}$$

Note that we now have a 1 on the left side and a fraction on the right side.

Multiplication by 1

Multiplying a value by 1 doesn't change the value. This means that if we multiply a term in an algebraic expression by

$$30 * \text{inch/pace}$$

we won't change the intrinsic value of the term, although we might change the units in which that value is expressed.

The term

$$30 * \text{inch/pace}$$

can be used to convert a quantity in units of paces to the same quantity in units of inches.

Move your body forward

So now we know that on the average, each time you execute one pace, you have moved your body forward by 30 inches. The next task is to determine how far you have moved your body when you have executed 2112 paces (the distance from home to school measured in paces).

Convert from paces to inches

We can begin by determining the distance from home to school in inches as follows:

$$\text{distance} = (2112 * \text{pace}) * (30 * \text{inch} / \text{pace})$$

A review of algebra

At this point, let's review a little algebra:

Given the expression: $((w/x) * (y/z))$

we can rewrite this as

$$(w * y) / (x * z)$$

Given this expression, we can reverse the process by rearranging and factoring out the terms producing our original expression:

$$((w/x) * (y/z))$$

An algebraic manipulation

We determined earlier that

$$\text{distance} = (2112 * \text{pace}) * (30 * \text{inch} / \text{pace})$$

Applying the algebraic process shown above to the problem at hand, we can rearrange terms and factor our expression to produce

$$\text{distance} = (2112 * 30 * \text{pace} * \text{inch}) / \text{pace}$$

The distance in inches

At this point, we have a fraction that contains the unit pace in both the numerator and the denominator. We can cancel those two terms leaving us with

$$\text{distance} = (2112 * 30 * \text{inch}) = 63360 * \text{inch}$$

So good so far. We now know the distance from home to school as expressed in units of inches (even though the original measurement was made in paces and not in inches). The distance hasn't changed. What has changed is the units in which we are expressing that distance.

Still not what we are looking for

While this is interesting, it still isn't what we are looking for. We want to know the distance in miles. Therefore, we need to convert the distance in inches to the distance in miles.

How many inches are in a mile?

At this point, we don't know how many inches are in a mile (but we could calculate it if we wanted to). However, we do know how many inches are in a foot and we know how many feet are in a mile.

A conversion factor for inches to feet

For starters, let's calculate a factor that we can use to convert from inches to feet. We know that

$$12 * \text{inches} = 1 * \text{foot}$$

If we divide each side by 12*inches, we get

$$1 = 1 \text{ foot} / 12 \text{ inches}$$

Let's refer to this as a conversion factor

Multiplication by 1

Since the expression on the right side is equal to 1, we can multiply any expression in an equation with the conversion factor

$$1 \text{ foot} / 12 \text{ inches}$$

without changing the intrinsic value of that expression. (Once again, although we won't be changing the intrinsic value of the expression, we may cause that intrinsic value to be expressed in different units.)

We can use this conversion factor to convert a distance value expressed in inches into the value for the same distance expressed in feet.

A conversion factor for feet to inches

We could also use the reciprocal of the expression to convert a distance value expressed in feet into the value for the same distance expressed in inches. I will have more to say on this later.

A conversion factor for feet to miles

Similarly, we could use the same procedure to show that

$$1 = 1 \text{ mile} / 5280 \text{ foot}$$

Again, since the expression on the right side is equal to 1, we can multiply any expression in an equation with the expression on the right without changing the intrinsic value of the original expression. This conversion expression can be used to convert from feet to miles. Its reciprocal could be used to convert from miles to feet.

Back to the problem at hand

We determined earlier that the distance expressed in inches from home to school is

$$\text{distance} = 63360 * \text{inch}$$

We can probably agree that multiplying the expression on the right as follows would not change the intrinsic value for the distance.

$$\text{distance} = 63360 * \text{inch} * 1 * 1$$

In fact, we could probably agree that it wouldn't change anything at all.

Substitute the two conversion factors

Having agreed on that, we can substitute the two conversion factors derived earlier, each of which has a value of 1, for the last two factors in our distance equation.

$$\text{distance} = 63360 * \text{inch} * (1 * \text{foot} / 12 * \text{inch}) * (1 * \text{mile} / 5280 * \text{foot})$$

Rearranging the terms

We can rearrange the terms and rewrite this equation as

$$\text{distance} = (63360 * \text{inch} * 1 * \text{foot} * 1 * \text{mile}) / (12 * \text{inch} * 5280 * \text{foot})$$

Canceling like terms

Canceling out like terms in the numerator and denominator leaves us with

$$\text{distance} = (63360 * \text{mile}) / (12 * 5280)$$

Doing the arithmetic, we get

$$\text{distance} = 1 * \text{mile}$$

(Obviously, I started out with a set of numbers that were designed to cause the final answer to be one mile, but that was simply for convenience.)

There you have it

You have seen a detailed procedure for converting from a distance expressed in paces (for a specific individual) to the same distance expressed in miles. Obviously, once you understand the overall procedure, you could omit and/or combine some of the steps to simplify the process.

Using JavaScript to convert units

Let's write, execute, and analyze a script that solves the same problem.

The objective of the script is to convert a distance of 2112 paces to a distance in miles where it is given that

- 100 paces = 3000 inches
- 12 inches = 1 foot
- 5280 feet = 1 mile

Please copy the code from [Listing 1](#) into an html file and open the file in your browser.

Listing 1 . Convert from paces to miles.

```
>!-- File JavaScript01.html -----  
----- >  
>html >>body >  
>script language="JavaScript1.3" >  
  
var d  = 2112    //distance, units = paces  
  
document.write(  
"d  = " + d  + " paces" + "<br/>")  
  
var d2 = 3000    //distance, units = inches  
var d3 = 100     //distance, units = paces
```

```
var f1 = d2/d3 //factor, units = inches/pace
document.write(
"f1 =" + f1 + " inches/pace" + "<br/>")
```

```
//pace*inch/pace = inch
var d = d *f1 //distance, units = inches
document.write(
"d = " + d + " inches" + "<br/>")
```

```
var f2 = 1/12 //factor, units = feet/inch
document.write(
"f2 = " + f2 + " feet/inch" + "<br/>")
```

```
//inch*feet/inch = feet
var d = d *f2 //distance, units = feet
document.write(
"d = " + d + " feet" + "<br/>")
```

```
var f3 = 1/5280 //factor, units = miles/foot
document.write(
"f3 = " + f3 + " miles/foot" + "<br/>")
```

```
//feet*mile/feet = mile
var d = d *f3 //distance, units = miles
document.write(
"d = " + d + " mile" + "<br/>")
```

```
//Now reverse the process using reciprocals
//((pace/inch)*(inch/foot)*(foot/mile) = pace/mile
var f4 =(1/f1)*(1/f2)*(1/f3) //factor, units =
paces/mile
document.write(
"f4 = " + f4 + " paces/mile" + "<br/>")
```

```
//miles*paces/mile = paces
var d = d *f4 //distance, units = paces
document.write(
```

```
"d = " + d + " paces" + "<br/>")
```

```
document.write("The End")
```

```
>/script >
```

```
>/body >>/html >
```

Screen output

When you open the html file in your browser, the text shown in [Figure 5](#) should appear in your browser window.

Figure 5 . Screen output for Listing #1.

```
d = 2112 paces
f1 =30 inches/pace
d = 63360 inches
f2 = 0.083333333333333333 feet/inch
d = 5280 feet
f3 = 0.0001893939393939394 miles/foot
d = 1 mile
f4 = 2112 paces/mile
d = 2112 paces
The End
```

Discuss the output first

Let's begin our discussion with the screen output shown in [Figure 5](#) and then go back and examine the code that produced that output.

[Figure 5](#) begins with a distance (labeled **d**) expressed in units of paces. The value is 2112 paces, which is the distance from your home to your school discussed earlier.

Convert paces to inches

Then [Figure 5](#) shows the conversion factor (labeled **f1**) that can be used to convert a value expressed in paces to a value expressed in inches.

That conversion factor is applied to the original distance producing an output consisting of the same distance expressed in inches instead of paces. The distance is now 63360 inches.

Convert from inches to feet

Following that, **f2** shows the conversion factor that can be used to convert a value expressed in inches to the same value expressed in feet.

That conversion factor is applied to the current distance value producing an output consisting of the same distance expressed in feet instead of inches. The distance is now 5280 feet.

Convert from feet to miles

Finally, [Figure 5](#) shows the conversion factor (labeled **f3**) that can be used to convert a value expressed in feet to the same value expressed in miles.

That conversion factor is applied to the current distance value producing a final output consisting of the same distance expressed in miles instead of feet. That distance is now one mile (obviously I chose the numbers to make it come out that way).

That is the number that we were looking for.

Reverse the process

If a conversion factor **X** can be used to convert from A-units to B-units, the reciprocal of **X** can be used to convert from B-units back to A-units.

As you will see later, the conversion factor **f4** shown in [Figure 5](#) was computed as the product of the reciprocals of **f1** , **f2** , and **f3** . The result is that the conversion factor labeled **f4** can be used to convert a value expressed in miles to the same value expressed in paces (according to the pace-length for one individual).

That conversion factor was applied to the distance expressed in miles to produce the same distance expressed in paces. As you might have guessed, this value of 2112 paces shown near the end of [Figure 5](#) matches the value shown at the beginning of [Figure 5](#).

Analysis of the script

Now lets analyze the script code in [Listing 1](#) that produced the text output shown in [Figure 5](#).

[Listing 1](#) begins by declaring and initializing the variables **d** , **d2** , and **d3** , which are respectively,

- The home to school distance in paces.
- The two values that represent the relationship between paces and inches for one individual.

Various values are displayed as the script executes using calls to the **document.write** method. You should have no difficulty identify the code that displays the values, so I will skip over that code in this discussion.

Compute, save, and apply the conversion factor f1

Then [Listing 1](#) declares a variable named **f1** and populates it with the value of a conversion factor that can be used to convert paces to inches. This conversion factor is computed from the known relationship between paces and inches mentioned earlier.

The conversion factor named **f1** is applied to the distance in paces converting it to a value that represents the same distance in inches (63360 inches).

Analysis of the units

The comment that reads

```
//pace*inch/pace = inch
```

is an analysis that shows the units that will result from applying the conversion factor to the distance at this point. As you can see, the pace terms will cancel and the result will be in inches.

Compute, save, and apply the conversion factor f2

Following that, a conversion factor named **f2** is computed that can be used to convert a value expressed in inches to the same value expressed in feet. This conversion factor is based on the known fact that there are 12 inches in a foot.

The factor named **f2** is applied to the distance that is now expressed in inches converting it to a new value that expresses the same distance in feet (5280 feet).

Analysis of the units

The comment that reads

```
//inch*feet/inch = feet
```

is an analysis that shows the units that will result from applying the conversion factor to the distance at this point. As you can see, the inch terms will cancel and the result will be in feet.

Compute, save, and apply the conversion factor f3

Following that, a conversion factor named **f3** is computed that can be used to convert a value expressed in feet to the same value expressed in miles. This conversion factor is based on the known fact that there are 5280 feet in a mile.

The factor named **f3** is applied to the distance that is now expressed in feet converting it to a new value that expresses the same distance in miles (1 mile).

Analysis of the units

The comment that reads

```
//feet*mile/feet = mile
```

is an analysis that shows the units that will result from applying the conversion factor to the distance at this point. As you can see, the feet terms will cancel and the result will be in miles.

That satisfies the specifications

That satisfies the original program specification. However, I mentioned earlier that if a conversion factor X can be used to convert from A-units to B-units, the reciprocal of X can be used to convert from B-units back to A-units.

Reversing the process

Continuing with [Listing 1](#), the comment that reads

```
 //(pace/inch)*(inch/foot)*(foot/mile) = pace/mile
```

shows the units that survive from a product of the reciprocals of **f1** , **f2** , and **f3** . As you can see, after canceling out inches and feet, the result of multiplying the reciprocals of those three conversion factors is a new conversion factor that can be used to convert a value expressed in miles to a new value that represents the same distance expressed in paces.

That conversion factor is applied to the distance in miles producing an output of 2112, which unsurprisingly, is the distance in paces that we started off with.

More substantive JavaScript examples

Assume that you drop a rock from a balloon at a height of 10000 feet above the ground, Further assume that the positive direction is up and the negative direction is down.

What would be the height of the rock above the ground at the end of ten seconds? What would be the height of the rock at the end of twenty seconds?

An equation relating distance, acceleration, and time

As you will learn in a future module, the following equation gives the distance that the rock will travel as a function of time assuming that the **initial velocity is zero** . (The assumption is that the rock was simply dropped **and not thrown** .)

$$d = 0.5 * g * t^2$$

where

- d is the distance traveled
- g is the acceleration of gravity (approximately -32.2 ft/s²)
- t is time in seconds

Free fall exercise

Please copy the code from [Listing 2](#) into an html file and open it in your browser.

Listing 2 . Free fall exercise.

Listing 2 . Free fall exercise.

```
>!-- File JavaScript02.html --<
>html<>body<
>script language="JavaScript1.3"<

//Function to compute free-fall distance for a
given time
// in seconds.
function distance(time){
    var g = -32.174; //acceleration of gravity
in feet/sec^2
    var d = 0.5 * g * time *
time;//(feet/sec^2)*sec^2 = feet
    return new Number(d.toFixed(0));
}//end function

//Compute and display height at ten seconds.
var t1 = 10;//time in seconds
var d1 = distance(t1);//distance traveled in
feet
var h1 = 10000 + d1;//height in feet

document.write("At " + t1 + " seconds:" +
<br/>");
document.write("distance traveled = " + d1 +
feet<br/>")
document.write("height = " + h1 + " feet<br/>")

//Compute and display height at twenty
seconds.
var t2 = 20;//time in seconds
var d2 = distance(t2);//distance traveled in
feet
var h2 = 10000 + d2;//height in feet

document.write("<br/>At " + t2 + " seconds:"
```

Listing 2 . Free fall exercise.

```
+"<br/>");  
document.write("distance traveled = " + d2 +"  
feet<br/>")  
document.write("height = " + h2 +" feet<br/>")  
  
document.write("<br/>The End")  
  
>/script</body></html<
```

Screen output

When you open the html file in your browser, the text shown in [Figure 6](#) should appear in your browser window.

Figure 6 . Screen output for Listing #2.

```
At 10 seconds:  
distance traveled = -1609 feet  
height = 8391 feet
```

```
At 20 seconds:  
distance traveled = -6435 feet  
height = 3565 feet
```

```
The End
```

As you can see from [Figure 6](#), after ten seconds, the rock will have traveled a distance of -1609 feet. (The minus sign indicates downward motion.)

Adding this value to the initial height of 10000 feet produces a value of 8391 for the height of the rock at the end of 10 seconds.

Similarly, [Figure 6](#) gives us 3565 for the height of the rock at 20 seconds.

Analysis of the code

The code in [Listing 2](#) begins by defining a function that computes and returns the distance traveled for a rock falling toward the earth for a given time interval since release assuming that the initial velocity was zero (the rock was simply dropped).

This function implements the [equation](#) shown earlier, and expects to receive the time as an input parameter. It returns the distance traveled during that time interval.

Note that the interaction of units is shown in the comments with the result that the returned value is in feet.

Call the function twice in succession

Then the code calls that function twice in succession for two different time intervals (10 seconds and 20 seconds) to determine the distance traveled during each time interval.

In both cases, the distance traveled is added to the initial height of 10000 feet to determine the height of the rock at the end of the time interval.

Also, in both cases, the time interval, the distance traveled, and the resulting height of the rock above the ground are displayed as shown in [Figure 6](#).

Note again that in all cases, the interactions of the various units are shown in the comments.

A plotting exercise

Let's do another exercise and this time plot the results.

Please prepare a piece of graph paper to plot a curve defined by several points. Interpret the grid lines such that you can plot values ranging from 0 to 10000 feet on the vertical axis and you can plot values ranging from 0 to 30 seconds on the horizontal axis.

JavaScript code

Copy the code from [Listing 3](#) into an html file and open it in your browser.

Listing 3 . A plotting exercise.

```
<!-- File JavaScript03.html -->
<html<<body<
<script language="JavaScript1.3"<

//Function to compute free-fall distance for a
given time
// interval in seconds.
function distance(time){
    var g = -32.174; //acceleration of gravity
in feet/sec^2
    var d = 0.5 * g * time *
time; //(feet/sec^2)*sec^2 = feet
    return new Number(d.toFixed(0));
} //end function

//Compute the height of the rock every two
seconds from
```

Listing 3 . A plotting exercise.

```
// release until the rock hits the ground.
var t = 0;//seconds
while(t <= 30){
    d = distance(t);//distance traveled in feet
    h = 10000 + d;//height in feet

    //Don't allow the height to go negative
    (underground)
    if(h < 0){
        h = 0;
    }//end if

    //Display time and height at current time.
    document.write("time = " + t +
        " height = " + h + "<br/<");

    t = t + 2;
} //end while

document.write("<br/>The End")

</script></body></html>
```

The screen output

When you open the html file in your browser, the text shown in [Figure 7](#) should appear in your browser window.

Figure 7 . Screen output for Listing #3.

Figure 7 . Screen output for Listing #3.

```
time = 0 height = 10000
time = 2 height = 9936
time = 4 height = 9743
time = 6 height = 9421
time = 8 height = 8970
time = 10 height = 8391
time = 12 height = 7683
time = 14 height = 6847
time = 16 height = 5882
time = 18 height = 4788
time = 20 height = 3565
time = 22 height = 2214
time = 24 height = 734
time = 26 height = 0
time = 28 height = 0
time = 30 height = 0
```

The End

Plot the data

Plot the 16 points shown in [Figure 7](#). Your horizontal axis should be the time axis and your vertical axis should be the height axis.

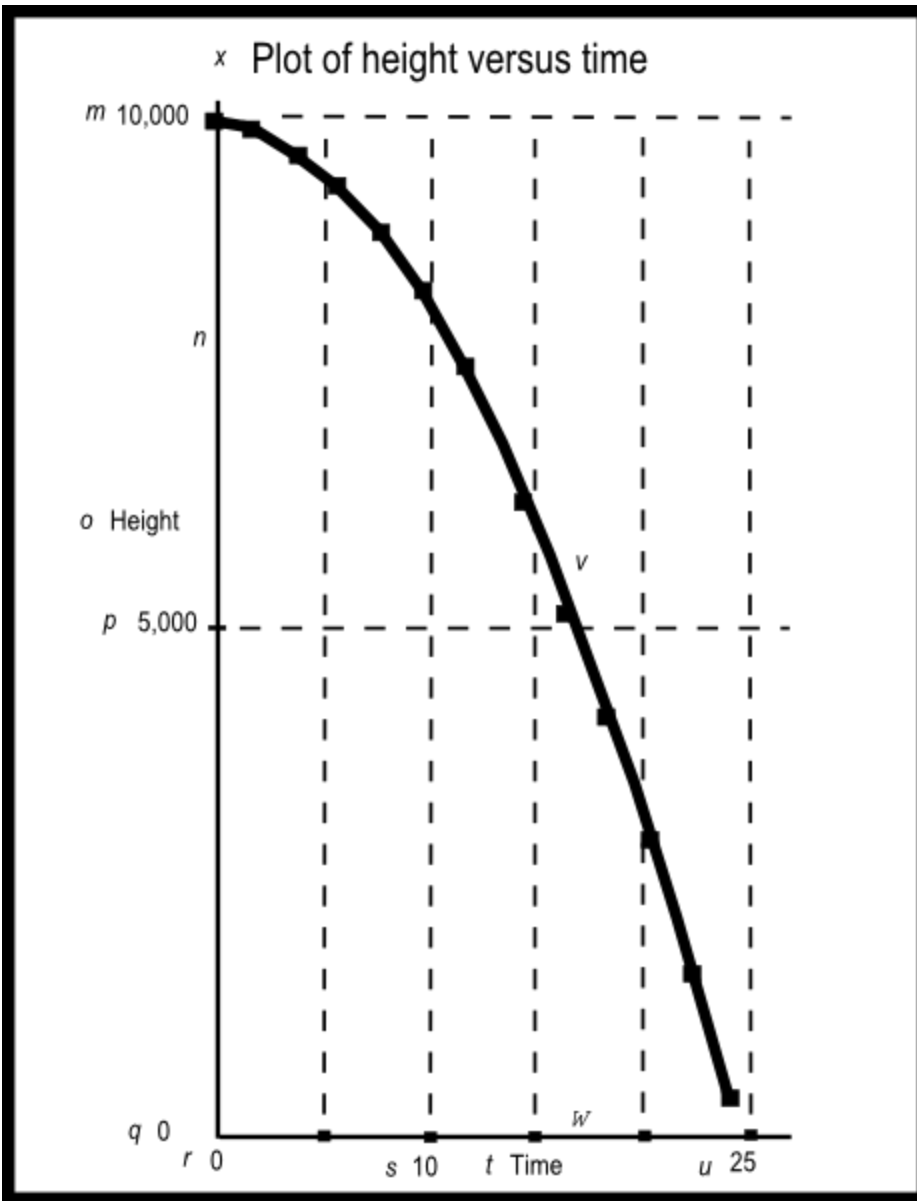
If we have both done everything correctly, the shape of your curve should be an upside-down parabola with its high point at a value of 10000. (If the word parabola is new to you, don't worry about it. It's not important for the purpose of this module.)

The graph

[Figure 8](#) shows an image of the graph that describes the trajectory of the falling rock. As usual, please ignore the lower-case letters scattered

throughout the image. They were placed there when the image was used for a different purpose earlier.

Figure 8 - The trajectory of the falling rock.



Change in height increases with time

Getting back to the height versus time values shown in [Figure 7](#) and plotted in [Figure 8](#), the value for height should decrease for each successive time value.

Also, the *change* in height should increase for each successive time interval until the height goes to zero. The height goes to zero when the rock has landed on the ground somewhere between 24 and 26 seconds. It continues at zero after that.

(Note that I put code into the script to prevent the value of height from going negative.)

The increase in the *change* in the height value during each successive time interval is the result of the time being squared in the equation given [earlier](#). It is also the result of the acceleration of gravity that causes the downward speed of the rock to increase as the rock falls.

Analysis of the code

[Listing 3](#) calls a function copied from the previous exercise inside a **while** loop to compute the distance traveled and the resulting height for a series of time values ranging from 0 to 30 seconds in two-second intervals.

Those values are displayed as shown in [Figure 7](#).

Understanding the code in [Listing 2](#) and [Listing 3](#) is important. However, it is more important at this point that you understand the handling of units as shown in the comments in the code.

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of .html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0350 Units and Dimensional Analysis
- File: Game0350.htm
- Published: 10/12/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0360 Motion -- Displacement and Vectors

The purpose of this module is to introduce motion and to explain displacement and vector analysis.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [Displacement](#)
 - [Addition and subtraction of vectors](#)
 - [Mathematical solutions](#)
 - [Using the Math.atan method](#)
 - [Mathematical solution to an earlier problem](#)
 - [Subtracting vectors](#)
 - [Adding more than two vectors](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME](#)

[2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

The purpose of this module is to introduce motion and to explain displacement and vector analysis.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). A graphic showing displacement.
- [Figure 2](#). Vector diagram for a familiar example.
- [Figure 3](#). A triangle with sides of 30, 40, and ?.
- [Figure 4](#). Vector diagram for the sum of six vectors.
- [Figure 5](#). Illustration of the parallelogram law.
- [Figure 6](#). Screen output for Listing #1.
- [Figure 7](#). Screen output for Listing #2.
- [Figure 8](#). Screen output for Listing #3.

Listings

- [Listing 1](#). Using the Math.atan method.
- [Listing 2](#). The sum of two vectors.
- [Listing 3](#). The sum of four vectors.

General background information

Bodies - not a reference to people

To begin with, physics textbooks often refer to something called *bodies* . By this, they are not referring to living bodies or the carcasses of once living creatures. Instead, they are referring to what people might ordinarily call objects.

For example, somewhere in this series of modules, we will discuss something called *a free body diagram* . A free body diagram can be drawn relative to any object, such as a child sitting motionless in a swing on the playground or a piano hanging on a rope.

Rest and motion

While nothing in the universe is truly motionless (at rest), some things around us such as buildings seem to be at rest, while other things such as planes, trains, and busses appear to be in motion.

There is no such thing as absolute rest. However, two bodies moving together at the same rate (with reference to some third body) may be deemed to be at rest relative to one another.

Therefore, if you are standing motionless on the Earth, you are at rest relative to the Earth with reference to the Sun.

Kinematics and kinetics

The *fundamental properties of motion* are often referred to as **kinematics** .

According to the course description, in addition to "*applications of mathematics and science*" , the course is also to include "*the utilization of matrix and vector operations, kinematics, and Newtonian principles in games and simulations*" ,

The relationships between motion and force are often referred to as **kinetics** . These are the topics that will be covered in this and the next several modules in this collection.

Displacement

If one body is moved with respect to another body, either one of them may be said to have experienced a **displacement** .

Displacement, along with vector analysis will be the primary topics covered in this module.

Discussion and sample code

Displacement

As mentioned above, if one body is moved with respect to another body, either one of them may be said to have experienced a **displacement** .

Displacement is not the same as distance

Assume, for example, that you go to the local high school and run five laps around the track, stopping exactly where you started. The distance that you have run will be the length of the path that you followed while running. However, while you may be completely out of breath by that point in time, the magnitude of your displacement will have been zero.

Scalars versus vectors

Distance is a **scalar** quantity, while displacement is a **vector** quantity. Scalar quantities have only magnitude, such as three kilometers.

Vector quantities have two parts: **magnitude** and **direction** . For example, you might describe a vector quantity as having a magnitude of three kilometers and a direction of northwest.

Magnitude and direction

A displacement vector must have both magnitude and direction. For example, if you were to run about three-fourths of the way around the track, the distance that you ran would be the length of your path.

However, the magnitude of your displacement would be the straight-line distance from your starting point to your stopping point.

The direction of your displacement would be an angle measured between some reference (such as due east, for example) and an imaginary line connecting your starting point and your stopping point.

If you stop where you start...

Getting back to the original proposition, if your stopping point is exactly the same as your starting point, the magnitude of your displacement vector will be zero and the angle of your displacement vector will be indeterminate.

Vector notation

Assume that there are three spots marked on the top of a table as A, B, and C. If an object is moved from A to B and then from B to C, using vectors, it may be stated that:

$$\text{vecAC} = \text{vecAB} + \text{vecBC}$$

where vecAB , vecBC , and vecAC represent vectors.

Typical notation

A typical notation for a vector in a physics textbook is something like \overrightarrow{AB} , possibly in boldface and/or italics, with a small arrow drawn above the two letters. However, there is no such small arrow on a QWERTY keyboard, making it very difficult for me to reproduce in these modules.

My alternative notation

Therefore, in these physics modules, I will use the following notation

vecAB

to indicate a vector extending from the spot labeled A to the spot labeled B.

In order to determine direction in an unambiguous way, you often need to know where the vector starts (the tail) and where it ends (the head). The notation used above indicates that the tail of the vector is at A and the head is at B.

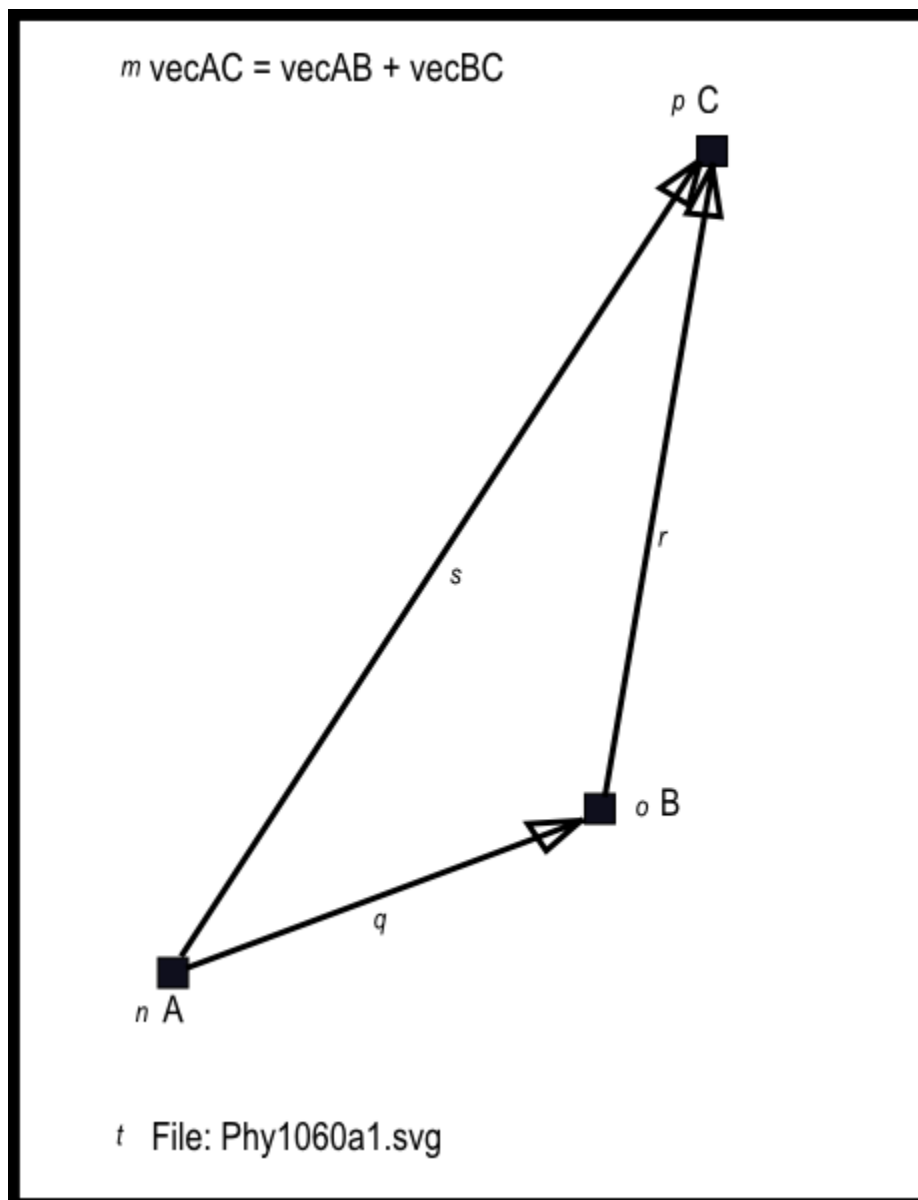
In some cases, particularly in JavaScript code where a vector is defined by magnitude and direction instead of by a starting point and an ending point, I will refer to a vector simply as

vecA, vecB, etc.

A graphic illustrating displacement

[Figure 1](#) is a graphic that illustrates displacement.

Figure 1 - A graphic showing displacement.



The triangle shown in [Figure 1](#) is a physical representation of the vector equation

$$\text{vecAC} = \text{vecAB} + \text{vecBC}$$

As usual, please ignore the lower-case letters scattered throughout the image, which were placed there for a purpose other than for use in this module.

What does this mean?

This vector equation does not mean that the algebraic sum of two sides of the triangle are equal to the third side. However, it does mean that the displacement vecAB together with the displacement vecBC , have an effect *that is equivalent* to the single displacement vector vecAC .

Scalar addition and subtraction

When adding or subtracting scalars, you only need to take the magnitude values of the scalars into account and perform normal arithmetic using those magnitude values. An example would be to add 6 birds and 5 birds to produce a sum of 11 birds.

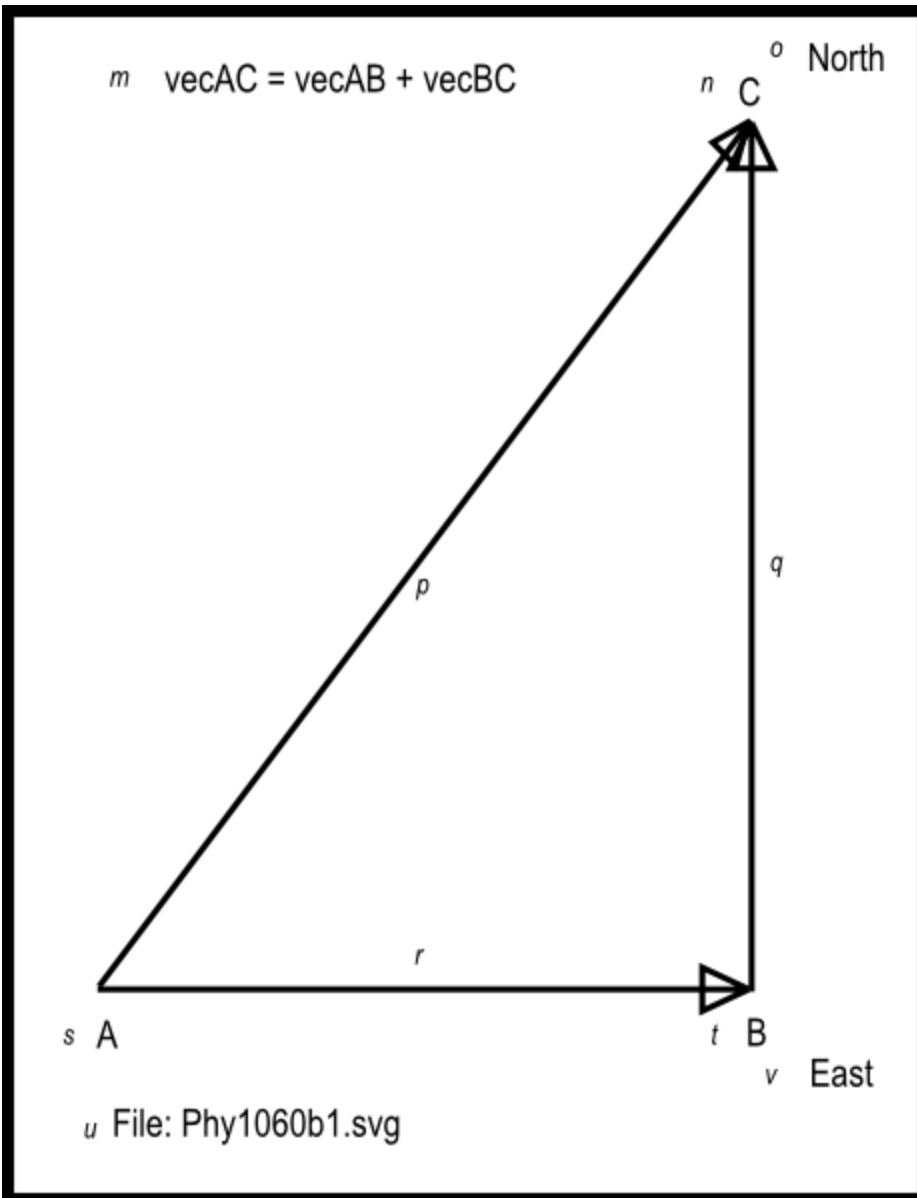
Vector addition and subtraction

However, when adding or subtracting vectors, you not only need to take the magnitude of the vectors into account, you also need to take the directions of the vectors into account.

A thought exercise

[Figure 2](#) shows the vector diagram for a familiar example.

Figure 2 - Vector diagram for a familiar example.



Pretend that you are standing at point A in [Figure 2](#). Then pretend that you walk 30 meters due east and designate that location as point B.

Up to this point, your displacement can be represented by \vec{AB} with a magnitude of 30 and an angle of zero (assuming that due east represents an angle of zero).

Walk 40 meters due north

Now pretend that you walk 40 meters due north and mentally designate the stopping point as C. Your displacement relative to point B can be

represented as vecBC with a magnitude of 40 meters and a direction of north or 90 degrees.

Displacement relative to point A

The interesting question is, what is your displacement relative to your starting point at A?

The displacement vector vecAC is the straight line from point A to point C, pointing in a direction from A to C. This vector is shown in the vector equation

$$\text{vecAC} = \text{vecAB} + \text{vecBC}$$

as the sum of the two vectors that describe the two segments of your journey from A to C.

Graphical addition of vectors

This process of connecting two or more vectors tail-to-head in sequence and then measuring the distance and direction from the tail of the first vector to the head of the last vector is known as the *graphical addition of vectors*. The resulting vector (in this case vecAC) is known as the sum of the vectors that make up the sequence. (The resulting vector is often called the resultant.)

A 3-4-5 triangle

Recalling what we learned in an earlier module where I discussed a 3-4-5 triangle, the magnitude of the vector from A to C (the hypotenuse of a right triangle with sides of 30 and 40) is 50 meters. You may also recall that the angle is approximately 53 degrees relative to the horizontal (east-west) line.

You should be able to measure the length of vecAC as well as the angle between that vector and the horizontal in [Figure 2](#) and get reasonable agreement with the above.

No requirement for a right triangle

While this example is based on a right triangle, that is not a requirement for adding vectors. I chose a 3-4-5 right triangle for the example because I knew what the answer would be in advance and also because it should have been familiar to you.

Graphic representation of vectors

Many problems in kinematics and kinetics can be solved graphically using a graphical representation of vectors.

Vectors are typically drawn as a heavy line between two points with an arrow head at one end and nothing in particular at the other end. The end with the arrow head is commonly called the head of the vector and the other end is commonly called the tail.

Somewhat inconvenient

Drawing vectors on graph paper can be inconvenient, particularly if you don't have graph paper available. Fortunately, as you will soon learn, it isn't necessary to use graphics to solve vector problems. You can also solve such problems mathematically, which will often be the better choice.

Addition and subtraction of vectors

There are at least two kinds of quantities in physics:

- **scalars** , having magnitude only
- **vectors** , having both magnitude and direction

You already know how to do scalar arithmetic. Otherwise, you probably wouldn't be interested in physics.

Another example

Let's go back to our original equation

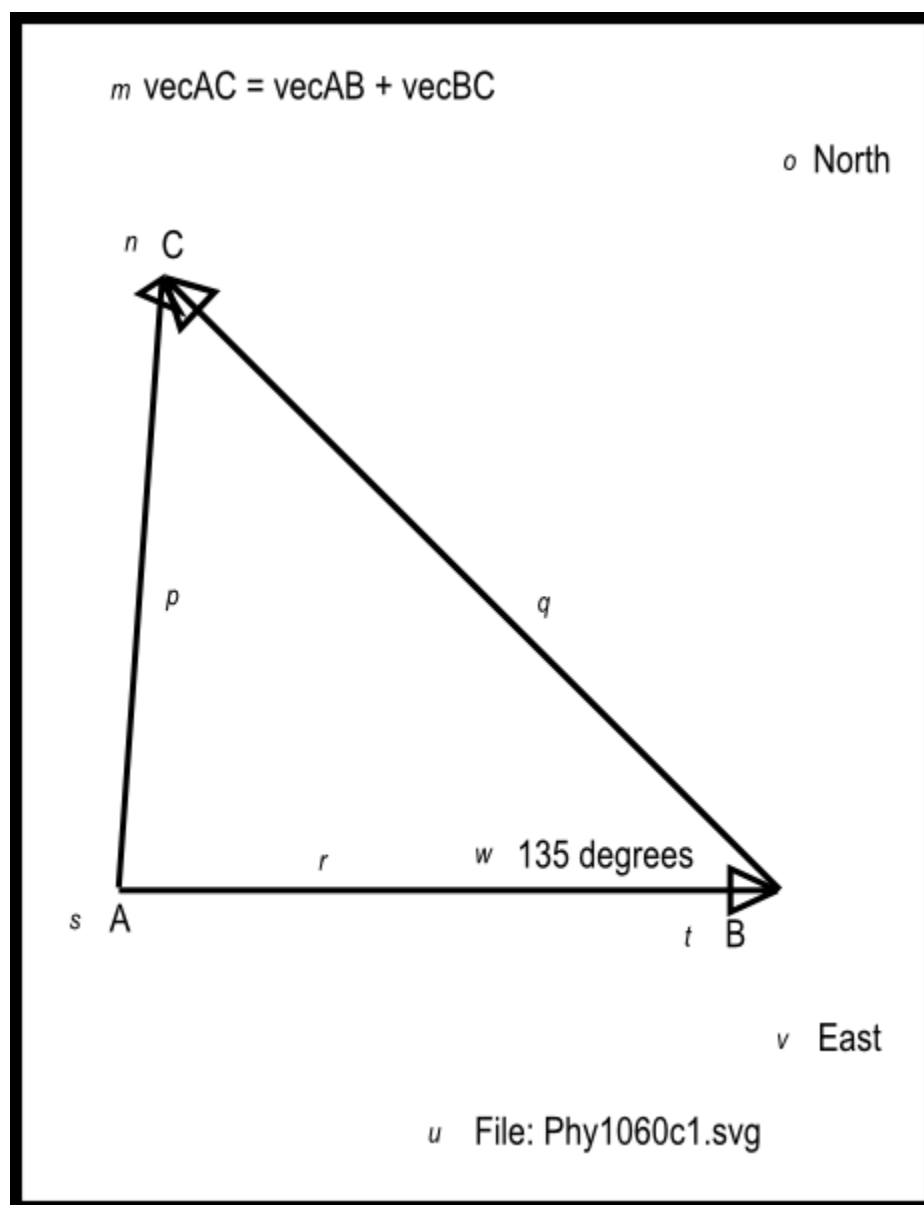
$$\text{vecAC} = \text{vecAB} + \text{vecBC}$$

and assume that the magnitude of vecAB is 30 and the magnitude of vecBC is 40. The sum of those two vectors (vecAC) can have a magnitude ranging anywhere from 10 to 70 depending on the relative angles of vecAB and vecBC .

A triangle with sides of 30, 40, and ?

Consider the triangle shown in [Figure 3](#).

Figure 3 - A triangle with sides of 30, 40, and ?.



Pretend that instead of walking due north from point B as shown in [Figure 2](#), you change direction and walk northwest (135 degrees relative to the east-west horizontal line with east being zero degrees) keeping the length at 40 meters.

What happened to the displacement?

What is the displacement of the point C relative to the point A? I can't do the arithmetic in my head for this problem, but I can measure the length of vecAC to be about 28 meters and the angle of vecAC to be a little less than 90 degrees. (I will show you how to write a script to solve this problem mathematically later.)

Any number of displacements can be added

There is no limit to the number of displacement vectors that can be added in this manner. For example, pretend that you walk from point A,

- 10 meters east to point B,
- 12 meters southwest to point C,
- 30 meters north to point D,
- 15 meters east to point E,
- 35 meters southwest to point F, and
- 44 meters north to point G where you stop.

Your displacement vecAG will be

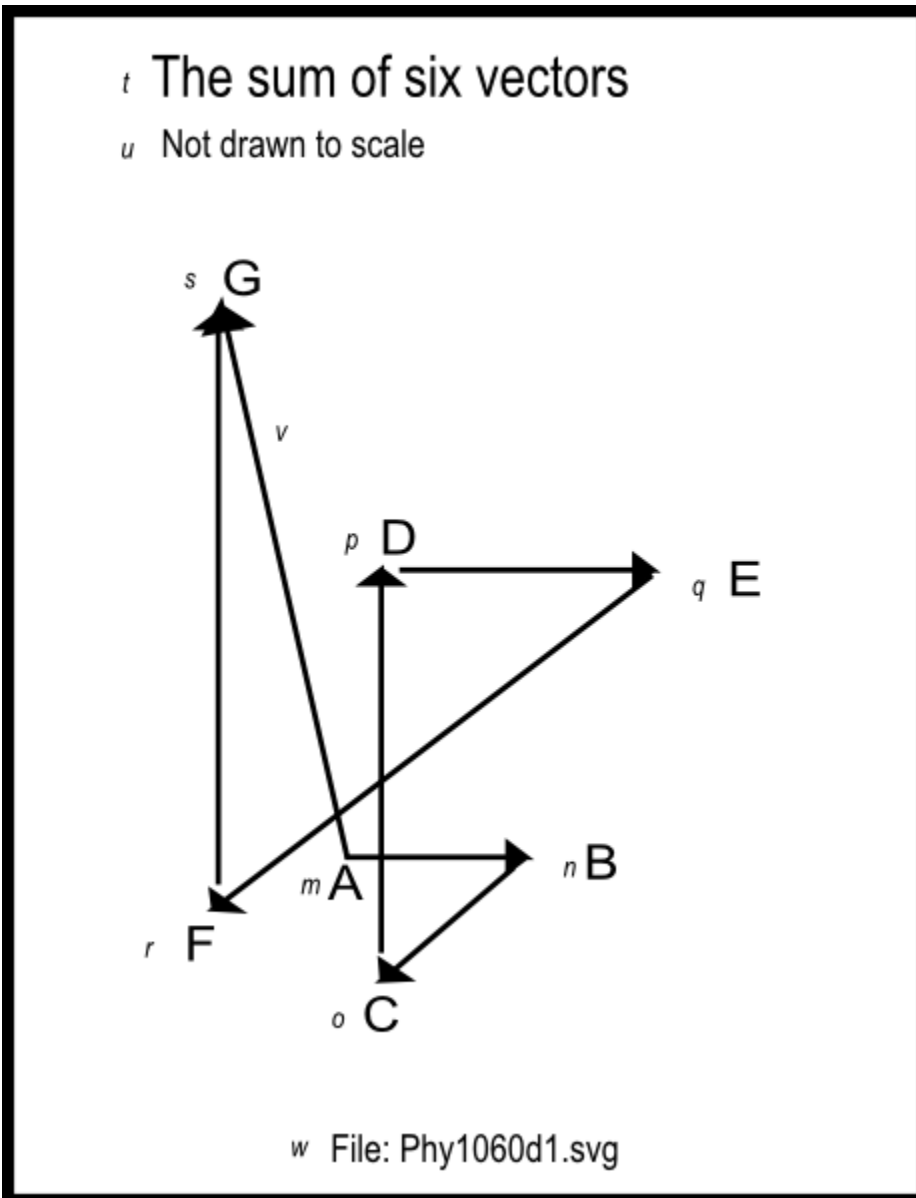
$$\text{vecAG} = \text{vecAB} + \text{vecBC} + \text{vecCD} + \text{vecDE} + \text{vecEF} + \text{vecFG}$$

even if your zigzag path crosses back over itself one or more times.

Vector diagram for the sum of six vectors

[Figure 4](#) shows the graphical addition of the six vectors described above.

Figure 4 - Vector diagram for the sum of six vectors.



The displacement in [Figure 4](#) is the vector with its tail at A and its head at G, which you could measure with a measuring stick and a protractor.

Can be extended to three (or more) dimensions

The physics textbook titled College Physics by Mendenhall, Eve, Keys, and Sutton contains the following example.

"If a man climbs up the mast of a ship sailing east while the tide carries it south, then that sailor may have three displacements at right angles, vecAB 30 feet upward, vecBC 100 feet eastward, vecCD 20 feet southward, all with

respect to the bed of the ocean. The three displacements vecAB , vecBC , and vecCD are equivalent to the single displacement vecAD ; and in the same way, any number of displacements may be added together."

Difficult to draw

Of course, we can't easily draw that 3D vector diagram on a flat sheet of paper or construct it on a flat computer monitor, but we can solve for the displacement vecAD if we are familiar with computations in three dimensions. (An explanation of 3D computations is beyond the scope of this module.)

The parallelogram law

There is a law that states:

The sum of two vectors in a plane is represented by the diagonal of a parallelogram whose adjacent sides represent the two vector quantities.

The resultant

As I mentioned earlier, the sum of two or more vectors is called their **resultant** .

In general, the resultant is not simply the algebraic or arithmetic sum of the vector magnitudes. Instead, in our original vector equation

$$\text{vecAC} = \text{vecAB} + \text{vecBC}$$

vecAC is the resultant of the sum of vecAB and vecBC in the sense that a single vector vecAC would have the same effect as vecAB and vecAD acting jointly.

Geometrical addition

The *parallelogram law* is a form of geometrical addition. In engineering and physics, it is often used to find graphical solutions to problems involving forces, velocities, displacements, accelerations, electric fields,

and other *directed* quantities. (Directed quantities have both magnitude and direction.)

Solving a vector problem with a parallelogram

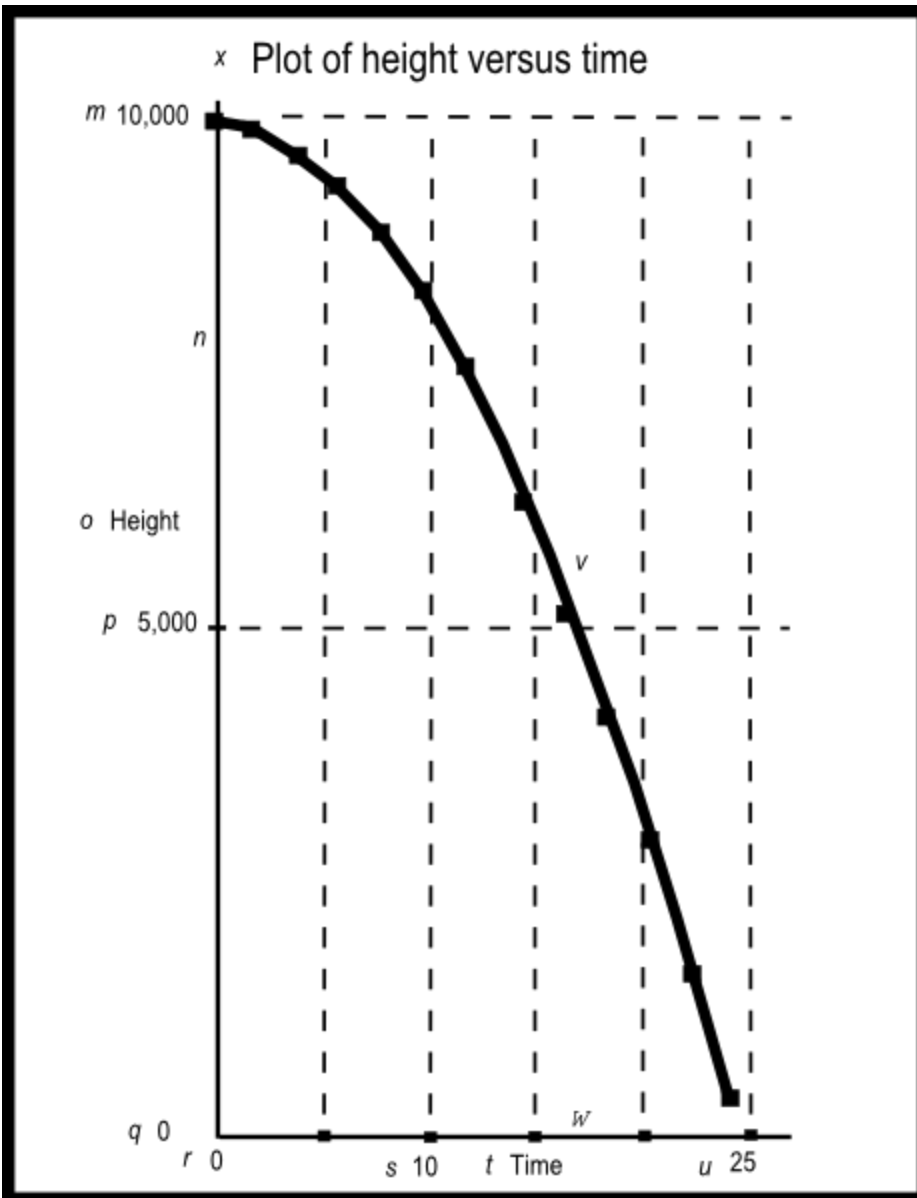
Pretend that you walk from point A

- 5 meters east to point B, followed by
- 6 meters at an angle of 30 degrees (north of east) to point C.

Let's use a vector diagram and the parallelogram law to find the resultant displacement vector \vec{AC} .

[Figure 5](#) illustrates the use of the parallelogram law to find a graphical solution to this problem.

Figure 5 - Illustration of the parallelogram law.



Put both vector tails at point A

Mentally designate the starting point on your vector diagram as point A. Construct a vector with a length of 5 meters and an angle of 0 degrees relative to the horizontal with its tail at point A. Mentally designate this vector as vecAB .

Construct a second vector with a length of 6 meters at an angle of 30 degrees with its tail at point A. Mentally designate this vector as vecBC .

Parallel lines

Construct a line at least 6 meters long starting at the head of vecBC . Make this line parallel to vecAB .

(There are drawing tools that make it easy to draw a line that is parallel to another line. This is one reason why this is a popular technique for graphically adding vectors.)

Construct another line, at least 7 meters long, starting at the head of vecAB . Make this line parallel to vecBC .

A parallelogram

Note the point where the two lines intersect. The two lines and the two vectors should now form a parallelogram as shown in [Figure 5](#).

Solving for the resultant vector

According to the parallelogram law, the resultant vector, vecAC , is represented by a directed line that extends from point A to the intersection of the two lines that you constructed earlier. In other words, the sum of the two vectors is equal to a diagonal line that extends from the starting point to the opposite corner on the parallelogram that you constructed.

Subtracting vectors

In normal arithmetic subtraction, a **subtrahend** is subtracted from a **minuend** to find a **difference**.

To subtract a subtrahend vector from a minuend vector (in a plane), add 180 degrees to the direction of the subtrahend vector, causing it to point in the opposite direction, and add the modified subtrahend vector to the minuend vector.

If you subtract vecAB in the above example from vecBC , you should be able to show that the difference vector is represented by the other diagonal in the parallelogram. However, you will probably find it easier to do this mathematically rather than doing it graphically.

Mathematical solutions

The horizontal component of the sum of two or more vectors is the sum of the horizontal components of the vectors.

The vertical component of the sum of two or more vectors is the sum of the vertical components of the vectors.

The horizontal and vertical components

The horizontal and vertical components respectively of a vector vecAB are equal to

- $\text{vecABh} = \text{vecABmag} * \cos(\text{angle})$
- $\text{vecABv} = \text{vecABmag} * \sin(\text{angle})$

where

- angle is the angle that the vector makes relative to the horizontal axis.
- vecABh , vecABv , and vecABmag are respectively the horizontal component, the vertical component, and the magnitude of the vector vecAB

The magnitude of the resultant vector

Given the horizontal and vertical components of the resultant vector, the magnitude of the resultant vector can be found using the Pythagorean theorem as the square root of the sum of the squares of the horizontal and vertical components.

The angle of the resultant vector

The angle that the resultant vector makes with the horizontal axis is the arctangent (corrected for quadrant) of the ratio of the vertical component to the horizontal component.

Using the `Math.atan` method

We will need to deal with several issues that arise when using the **Math.atan** method. Therefore, I will write a script that contains a function to deal with those issues. Then we can simply copy that function into future scripts without having to consider those issues at that time.

Please copy the code shown in [Listing 1](#) into an html file and open the file in your browser.

Listing 1 . Using the Math.atan method.

```
<!-- File JavaScript01.html -->
<html><body>
<script language="JavaScript1.3">

//The purpose of this function is to receive
the adjacent
// and opposite side values for a right
triangle and to
// return the angle in degrees in the correct
quadrant.
function getAngle(x,y){
    if((x == 0) && (y == 0)){
        //Angle is indeterminate. Just return
zero.
        return 0;
    }else if((x == 0) && (y > 0)){
        //Avoid divide by zero denominator.
        return 90;
    }else if((x == 0) && (y < 0)){
        //Avoid divide by zero denominator.
        return -90;
    }else if((x < 0) && (y >= 0)){
```

Listing 1 . Using the Math.atan method.

```
//Correct to second quadrant
return Math.atan(y/x)*180/Math.PI + 180;
}else if((x < 0) && (y <= 0)){
    //Correct to third quadrant
    return Math.atan(y/x)*180/Math.PI + 180;
}else{
    //First and fourth quadrants. No
correction required.
    return Math.atan(y/x)*180/Math.PI;
} //end else
} //end function getAngle

//Test for a range of known angles.
var angIn = 0;
var angOut = 0;
var x = 0;
var y = 0;
while(angIn <= 360){
    x = Math.cos(angIn*Math.PI/180);
    y = Math.sin(angIn*Math.PI/180);
    angOut = getAngle(x,y).toFixed(1);
    document.write("angle = " + angOut + "
<br/>");
    angIn = angIn + 30;
} //end while loop

document.write("The End")

</script>
</body></html>
```

The screen output

The text shown in [Figure 6](#) should appear in your browser window when you open the html file in your browser.

Figure 6 . Screen output for Listing #1.

```
angle = 0.0
angle = 30.0
angle = 60.0
angle = 90.0
angle = 120.0
angle = 150.0
angle = 180.0
angle = 210.0
angle = 240.0
angle = 270.0
angle = -60.0
angle = -30.0
angle = -0.0
The End
```

Structure of the script

The script shown in [Listing 1](#) begins by defining a function named **getAngle** . The purpose of this function is to return an angle in degrees in the correct quadrant based on the lengths of the adjacent and opposite sides of an enclosing right triangle.

Then the script uses a **while** loop to test the function for a series of known angles ranging from 0 to 360 degrees inclusive.

An indeterminate result

The **getAngle** function calls the **Math.atan** method to compute the angle whose tangent is the ratio of the opposite side (y) to the adjacent side (x) of a right triangle.

If both x and y are zero, the ratio y/x is indeterminate and the value of the angle cannot be computed. In fact there is no angle corresponding to the ratio 0/0. However, the function must either return the value of an angle, or must return some sort of flag indicating that computation of the angle is not possible.

In this case, the function simply returns the value zero for the angle.

Avoiding division by zero

If the value of x is zero and the value of y is not zero, the ratio y/x is infinite. Therefore, the value of the angle cannot be computed. However, in this case, the angle is known to be 90 degrees (for y greater than zero) or 270 degrees (-90 degrees, for y less than zero). The **getAngle** function traps both of those cases and returns the correct angle in each case.

Correcting for the quadrant

The **Math.atan** method receives one parameter and it is either a positive or negative value. If the value is positive, the method returns an angle between 0 and 90 degrees. If the value is negative, the method returns an angle between 0 and -90 degrees. Thus, the angles returned by the **Math.atan** method always lie in the first or fourth quadrants.

(Actually, as I mentioned earlier, +90 degrees and -90 degrees are not possible because the tangent of +90 degrees or -90 degrees is an infinitely large positive or negative value. However, the method can handle angles that are very close to +90 or -90 degrees.)

A negative y/x ratio

If the y/x ratio is negative, this doesn't necessarily mean that the angle lies in the fourth quadrant. That negative ratio could result from a positive value

for y and a negative value for x. In that case, the angle would lie in the second quadrant between 90 degrees and 180 degrees.

The **getAngle** function tests the signs of the values for y and x. If the signs indicate that the angle lies in the second quadrant, the value returned from the **Math.atan** method is corrected to place the angle in the second quadrant. The corrected angle is returned by the **getAngle** function.

A positive y/x ratio

Similarly, if the y/x ratio is positive, this doesn't necessarily mean that the angle lies in the first quadrant. That positive ratio could result from a negative y value and a negative x value. In that case, the angle would lie in the third quadrant between 180 degrees and 270 degrees.

Again, the **getAngle** function tests the signs of the values for y and x. If both values are negative, the value returned from the **Math.atan** method is corrected to place the angle in the third quadrant.

No corrections required...

Finally, if no corrections are required for the quadrant, the **getAngle** function returns the value returned by the **Math.atan** method. Note however, that in all cases, the **Math.atan** method returns the angle in radians. That value is converted to degrees by the **getAngle** function and the returned value is in degrees.

Positive and negative angles

As you can see from the results of the test shown in [Figure 6](#), angles in the first, second, and third quadrants are returned as positive angles in degrees. However, angles in the fourth quadrant are returned as negative angles in degrees.

Mathematical solution to an earlier problem

[Earlier](#) in this module, I described a problem involving the sum of two displacement vectors. One vector had a magnitude of 30 meters with an angle of 0. The other vector had a magnitude of 40 meters with an angle of 135 degrees.

On the basis of a graphic solution, I estimated the length of the resultant vector to be about 28 meters and the angle of the resultant vector to be a little less than 90 degrees. I promised to provide a mathematical solution for that problem later, and that time has come.

Please copy the code shown in [Listing 2](#) into an html file and open that file in your browser.

Listing 2 . The sum of two vectors.

```
<!-- File JavaScript02.html -->
<html><body>
<script language="JavaScript1.3">

//The purpose of this function is to receive the
adjacent
// and opposite side values for a right triangle
and to
// return the angle in degrees in the correct
quadrant.
function getAngle(x,y){
    if((x == 0) && (y == 0)){
        //Angle is indeterminate. Just return zero.
        return 0;
    }else if((x == 0) && (y > 0)){
        //Avoid divide by zero denominator.
        return 90;
    }else if((x == 0) && (y < 0)){
        //Avoid divide by zero denominator.
        return -90;
    }else if((x < 0) && (y >= 0)){
        //Correct to second quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
```

```

    }else if((x < 0) && (y <= 0)){
        //Correct to third quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else{
        //First and fourth quadrants. No correction
        required.
        return Math.atan(y/x)*180/Math.PI;
    }//end else
}//end function getAngle

```

```

//Specify the magnitudes and angles of two
vectors.
//Convert the angles from degrees to radians.
var vecABmag = 30;//at an angle of 0
var vecABang =
0*Math.PI/180;//degrees*radians/degrees=radians

var vecBCmag = 40;//at an angle of 135
var vecBCang =
135*Math.PI/180;//degrees*radians/degrees=radians

```

```

//Compute the horizontal and vertical components
of each vector.
var vecABh = vecABmag*Math.cos(vecABang);
var vecABv = vecABmag*Math.sin(vecABang);

```

```

var vecBCh = vecBCmag*Math.cos(vecBCang);
var vecBCv = vecBCmag*Math.sin(vecBCang);

```

```

//Compute the sums of the horizontal and vertical
components from
// the two vectors to get the horizontal and
vertical component
// of the resultant vector.
var vecResultH = vecABh + vecBCh;

```

```

var vecResultV = vecABv + vecBCv;

//Use the Pythagorean theorem to compute the
magnitude of the
// resultant vector.
var vecResultMag =
Math.sqrt(Math.pow(vecResultH,2) +
           Math.pow(vecResultV,2));

//Compute the angle of the resultant.
vecResultAng = getAngle(vecResultH,vecResultV);

document.write("Hello from JavaScript" + "<br/>")
document.write("vecABv = " + vecABv + "<br/>");
document.write("vecABh = " + vecABh + "<br/>");
document.write("vecBCv = " + vecBCv + "<br/>");
document.write("vecBCh = " + vecBCh + "<br/>");
document.write("vecResultMag = " + vecResultMag +
" <br/>");
document.write("vecResultAng = " + vecResultAng +
" <br/>");

document.write("The End")

</script>
</body></html>

```

Screen output

When you open the html file in your browser, the text shown in [Figure 7](#) should appear in your browser window.

Figure 7 . Screen output for Listing #2.

```
Hello from JavaScript
vecABv = 0
vecABh = 30
vecBCv = 28.284271247461902
vecBCCh = -28.2842712474619
vecResultMag = 28.336261665087125
vecResultAng = 86.5286817554714
The End
```

Explanation of the code

[Listing 2](#) begins with a copy of the **getAngle** method that I explained in conjunction with [Listing 1](#). I won't repeat that explanation here.

Define magnitudes and angles of vectors

Then [Listing 2](#) declares and initializes four variables that represent the magnitude and angle of each of the two vectors. The variable names that end with "mag" contain magnitude values and the variable names that end with "ang" contain angle values, which are converted from degrees to radians.

Compute horizontal and vertical components

Then [Listing 2](#) calls the **Math.cos** and **Math.sin** methods to compute the horizontal and vertical components of each vector. The variable names that end with "h" contain horizontal component values and the variable names that end with "v" contain vertical component values.

Compute the sums of the components

The procedure for adding two or more vectors is to

- Add the horizontal components of all the vectors to get the horizontal component of the resultant vector.
- Add the vertical components of all the vectors to get the vertical component of the resultant vector.
- Compute the magnitude (if needed) of the resultant vector as the square root of the **sum of the squares** of the horizontal and vertical components.
- Compute the angle (if needed) of the resultant vector as the arctangent of the ratio of the vertical component and the horizontal component.

[Listing 2](#) computes the sum of the horizontal components from each of the vectors and saves the result in the variable named **vecResultH** . Then it computes the sum of the vertical components and saves the result in the variable named **vecResultV** .

Compute the magnitude of the resultant

In many cases, simply knowing the horizontal and vertical components is the solution to the problem. However, in some cases, it is necessary to convert those components into a magnitude and an angle.

There are a couple of ways to compute the magnitude. One way is to use the Pythagorean theorem as described [above](#) to compute the square root of the sum of the squares of the horizontal and vertical components.

The **Math.sqrt** and **Math.pow** methods

The method named **Math.sqrt** can be called to return the square root of its positive argument. (If the argument is negative, the method returns NaN, which is an abbreviation for "Not a Number".)

The method named **Math.pow** can be called to raise the value specified by the first argument to the power specified by the second argument.

[Listing 2](#) calls those two methods to compute the magnitude of the resultant and to save that value in the variable named **vecResultMag** .

Compute the angle of the resultant

Then [Listing 2](#) calls the **getAngle** method that we developed earlier to get the angle of the resultant and to save the angle in degrees in the variable named **vecResultAng** .

Display the results

Finally [Listing 2](#) calls the **document.write** method several times in succession to display the values stored in some of the variables mentioned above in the browser window as shown in [Figure 7](#). As you can see, the magnitude and angle of the resultant agrees with the estimate that I made using the graphic method earlier.

Subtracting vectors

To subtract one vector from another using the mathematical approach, simply reverse the algebraic sign on the horizontal and vertical components for the subtrahend vector and perform the addition shown in [Listing 2](#).

Adding more than two vectors

Let's work through a problem that requires the addition of four vectors.

A graphical solution

First I would like for you to lay out the vectors, tail-to-head on graph paper to create a solution to this problem.

Pretend that you take a walk consisting of the following sequential segments. Angles are specified relative to east with east being zero degrees. Positive angles are counter-clockwise and negative angles are clockwise.

The segments are as follows:

- 12 meters at 20 degrees
- 8 meters at 120 degrees
- 10.5 meters at -74 degrees

- 12.5 meters at 145 degrees

What is your final displacement?

When you reach the end of the walk, what is your displacement (distance and direction) relative to the location at which you began the walk. You should be able to do a reasonable job of measuring the displacement using your vector diagram.

A mathematical solution

Please copy the code shown in [Listing 3](#) into an html file and open the html file in your browser.

Listing 3 . The sum of four vectors.

```
<!-- File JavaScript03.html -->
<html><body>
<script language="JavaScript1.3">

//The purpose of this function is to receive the
adjacent
// and opposite side values for a right triangle
and to
// return the angle in degrees in the correct
quadrant.
function getAngle(x,y){
  if((x == 0) && (y == 0)){
    //Angle is indeterminate. Just return zero.
    return 0;
  }else if((x == 0) && (y > 0)){
    //Avoid divide by zero denominator.
    return 90;
  }else if((x == 0) && (y < 0)){
    //Avoid divide by zero denominator.
    return -90;
  }else if((x < 0) && (y >= 0)){
    //Correct to second quadrant
```



```

        return Math.atan(y/x)*180/Math.PI + 180;
    }else if((x < 0) && (y <= 0)){
        //Correct to third quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else{
        //First and fourth quadrants. No correction
        required.
        return Math.atan(y/x)*180/Math.PI;
    }//end else
}//end function getAngle

```

```

//Specify the magnitudes and angles of four
vectors.
//Convert the angles from degrees to radians.
var vecAmag = 12;
var vecAang =
20*Math.PI/180;//degrees*radians/degrees=radians

var vecBmag = 8;
var vecBang =
120*Math.PI/180;//degrees*radians/degrees=radians

var vecCmag = 10.5;
var vecCang =
-74*Math.PI/180;//degrees*radians/degrees=radians

var vecDmag = 12.5;
var vecDang =
145*Math.PI/180;//degrees*radians/degrees=radians

//Compute the horizontal and vertical components
of each vector.
var vecAh = vecAmag*Math.cos(vecAang);
var vecAv = vecAmag*Math.sin(vecAang);

```

```
var vecBh = vecBmag*Math.cos(vecBang);
var vecBv = vecBmag*Math.sin(vecBang);

var vecCh = vecCmag*Math.cos(vecCang);
var vecCv = vecCmag*Math.sin(vecCang);

var vecDh = vecDmag*Math.cos(vecDang);
var vecDv = vecDmag*Math.sin(vecDang);

//Compute the sums of the horizontal and vertical
components from
// the four vectors to get the horizontal and
vertical component
// of the resultant vector.
var vecResultH = vecAh + vecBh + vecCh + vecDh;
var vecResultV = vecAv + vecBv + vecCv + vecDv;

//Use the Pythagorean theorem to compute the
magnitude of the
// resultant vector.
var vecResultMag =
Math.sqrt(Math.pow(vecResultH,2) +
           Math.pow(vecResultV,2));

//Compute the angle of the resultant vector.
vecResultAng = getAngle(vecResultH,vecResultV);

document.write("Hello from JavaScript" + "<br/>")
document.write(
    "vecResultMag = " + vecResultMag.toFixed(2) + "
<br/>");
document.write(
    "vecResultAng = " + vecResultAng.toFixed(2) + "
<br/>");

document.write("The End")
```

```
</script>  
</body></html>
```

Screen output

The text shown in [Figure 8](#) should appear in your browser when you open the html file in your browser.

Figure 8 . Screen output for Listing #3.

```
Hello from JavaScript  
vecResultMag = 8.11  
vecResultAng = 90.49  
The End
```

As you can see, the displacement vector is just a little over eight meters at an angle of about 90.5 degrees. In other words, your ending position was approximately 8 meters north of your starting position. Hopefully you were able to arrive at a very similar answer using vector diagram.

Explanation of the code

The code in [Listing 3](#) is very similar to the code in [Listing 2](#). However, there is more of it because the code is adding four vectors instead of only 2.

As before, [Listing 3](#) begins with the **getAngle** method that we developed earlier.

Magnitude and angle for each vector

Then [Listing 3](#) defines eight variables containing the magnitudes and angles of the four segments of the walk.

(Note that the names that I used for the variables in this script are somewhat simpler than the names that I used in [Listing 2](#).)

Horizontal and vertical components

Following that, [Listing 3](#) computes the horizontal and vertical components for each of the four vectors and saves those values in eight different variables.

Sums of components

Then [Listing 3](#) computes and saves the sum of all four horizontal components and computes and saves the sum of all four vertical components.

After that, there is essentially no difference between the script in [Listing 3](#) and the script in [Listing 2](#).

Other vector operations

In addition to the addition and subtraction of vectors, there are other operations that can be performed using vectors, such as the dot product and the cross product. However, those operations are beyond the scope of this module.

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of .html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0360 Motion -- Displacement and Vectors
- File: Game0360.htm
- Published: 10/13/12
- Revised: 02/01/16

Note:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0370 Motion -- Uniform and Relative Velocity

The purpose of this module is to explain uniform velocity and relative velocity.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
- [Discussion and sample code](#)
 - [A simple exercise on uniform velocity](#)
 - [An exercise on average velocity](#)
 - [Multiple concurrent velocities](#)
 - [Exercise #1 for a man on a train](#)
 - [Exercise #2 for a man on a train](#)
 - [An exercise with three vectors in a plane](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

The purpose of this module is to explain uniform velocity and relative velocity.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Screen output for Listing #1.
- [Figure 2](#). Vector diagram for the hockey puck.
- [Figure 3](#). Screen output for Listing #2.
- [Figure 4](#). Vector diagram for a man on a train.
- [Figure 5](#). Screen output for Listing #3.
- [Figure 6](#). Another vector diagram for a man on a train
- [Figure 7](#). Screen output for Exercise #2 for a man on a train.
- [Figure 8](#). Vector diagram for man on aircraft carrier.
- [Figure 9](#). Screen output for Listing #4.

Listings

- [Listing 1](#). A simple exercise on uniform velocity.
- [Listing 2](#). An exercise on average velocity.
- [Listing 3](#). Exercise #1 for a man on a train.
- [Listing 4](#). An exercise with three vectors in a plane.

General background information

What is velocity?

Velocity is the *time rate of change of position* . Since displacement is the change of position, velocity is also the *rate of displacement* . Since displacement is a vector quantity, velocity is also a vector quantity -- it has magnitude and direction.

An airplane may traverse 350 miles in a northeasterly direction in one hour. This statement describes a vector quantity (velocity) because it has both magnitude and direction.

The same airplane may traverse 350 miles in a southwesterly direction in one hour. This statement describes a different velocity. Even though the magnitude is the same in both cases, the direction is different. Hence, the velocity is different.

What is speed?

Speed is *distance covered in a unit of time* without reference to direction. Since there is no reference to direction, speed is a scalar quantity.

A car may traverse 60 miles in one hour. An airplane may traverse 350 miles in one hour. An excellent runner may traverse one mile in three minutes. All of these statements describe scalar quantities (speed) since there is no reference to direction.

A person in a seat on a Ferris wheel may travel around the axle in the center of the wheel at a constant rotational speed. However, that person's velocity is not constant. The velocity is continually changing because the direction of motion is continually changing.

Uniform velocity

Velocity is uniform when equal distances along a straight line are traversed in equal intervals of time. In this case

$$v = s/t$$

where

- v is uniform velocity

- s is the distance traveled (displacement)
- t is an interval of time

What about the direction?

Because uniform velocity occurs along a straight line, the computations involving uniform velocity, distance, and time may not necessarily include the direction of that straight line. However, since both displacement and velocity are vector quantities, it is understood that the direction of velocity is the same as the direction of the displacement.

On the other hand, as you will see later, when combining two or more uniform velocities into a single resultant velocity, the directions of the different uniform velocities becomes extremely important.

Speed, distance, and time

Note also that this same equation can be applied to computations involving speed, distance, and time where

- v represents speed and not velocity
- s is the distance traveled (not necessarily in a straight line)
- t is an interval of time

Units of velocity

The units of velocity are units of distance (such as meters) divided by units of time (such as seconds).

Discussion and sample code

A simple exercise on uniform velocity

Let's begin this section with a short exercise involving uniform velocity.

How long does it take an airplane traveling at a uniform velocity of 350 miles per hour to traverse a distance of 700 miles?

Please copy the code shown in [Listing 1](#) into an html file and open the file in your browser.

Listing 1 . A simple exercise on uniform velocity.

```
<!------- File JavaScript01.html ----  
----->  
<html><body>  
<script language="JavaScript1.3">  
  
document.write("Start Script" + "<br/>")  
  
var velocity = 350;//uniform velocity in  
miles/hour  
var distance = 700;//miles  
  
var time =  
distance/velocity;//miles/(miles/hour) = hours  
  
document.write("uniform velocity = " +  
velocity +  
           " miles/hour<br/>");  
document.write("distance = " + distance + "  
miles<br/>");  
document.write("time = " + time + "  
hours<br/>");  
document.write("End Script")  
  
</script>  
</body></html>
```

Screen output

The text shown in [Figure 1](#) should appear in your browser window when you open the html file in your browser.

Figure 1 . Screen output for Listing #1.

```
Start Script
speed = 350 miles/hour
distance = 700 miles
time = 2 hours
End Script
```

Analysis of the code

The equation given [earlier](#) that shows the relationship among uniform velocity, distance, and time can be manipulated algebraically to find any one of the three terms when the values of the other two terms are known. The objective in this exercise is to find the time required to traverse a known distance with a known uniform velocity.

Algebraic manipulation of the equation

Multiplying both sides of the equation by t and dividing both sides by v allows us to rewrite the equation as

$$t = s / v$$

where

- v is uniform velocity

- s is the distance traveled
- t is an interval of time

The solution to the problem is...

This version of the equation is used in [Listing 1](#) to solve for the time required to traverse 700 miles at a uniform velocity of 350 miles/hour. The answer is 2 hours as shown in [Figure 1](#).

Declare and initialize variables

The code in [Listing 1](#) begins by declaring and initializing variables named **velocity** and **distance** . Note that the units are shown in the comments. I recommend that you always do that in order to keep the units straight.

Perform the arithmetic

Then [Listing 1](#) declares a variable named **time** and sets its value equal to **distance** divided by **velocity** . Once again, note the treatment of the units in the comments, showing that the units for the result of the division are hours.

Display the results

Finally, [Listing 1](#) calls the **document.write** method several times in succession to display the results shown in [Figure 1](#).

An exercise on average velocity

A hockey puck is struck by a player causing it to travel northeast at a uniform velocity of 50 feet per second for 15 feet, at which point it is struck by another player.

When struck by the second player, the direction of motion is changed to northwest with a uniform velocity of 60 feet per second.

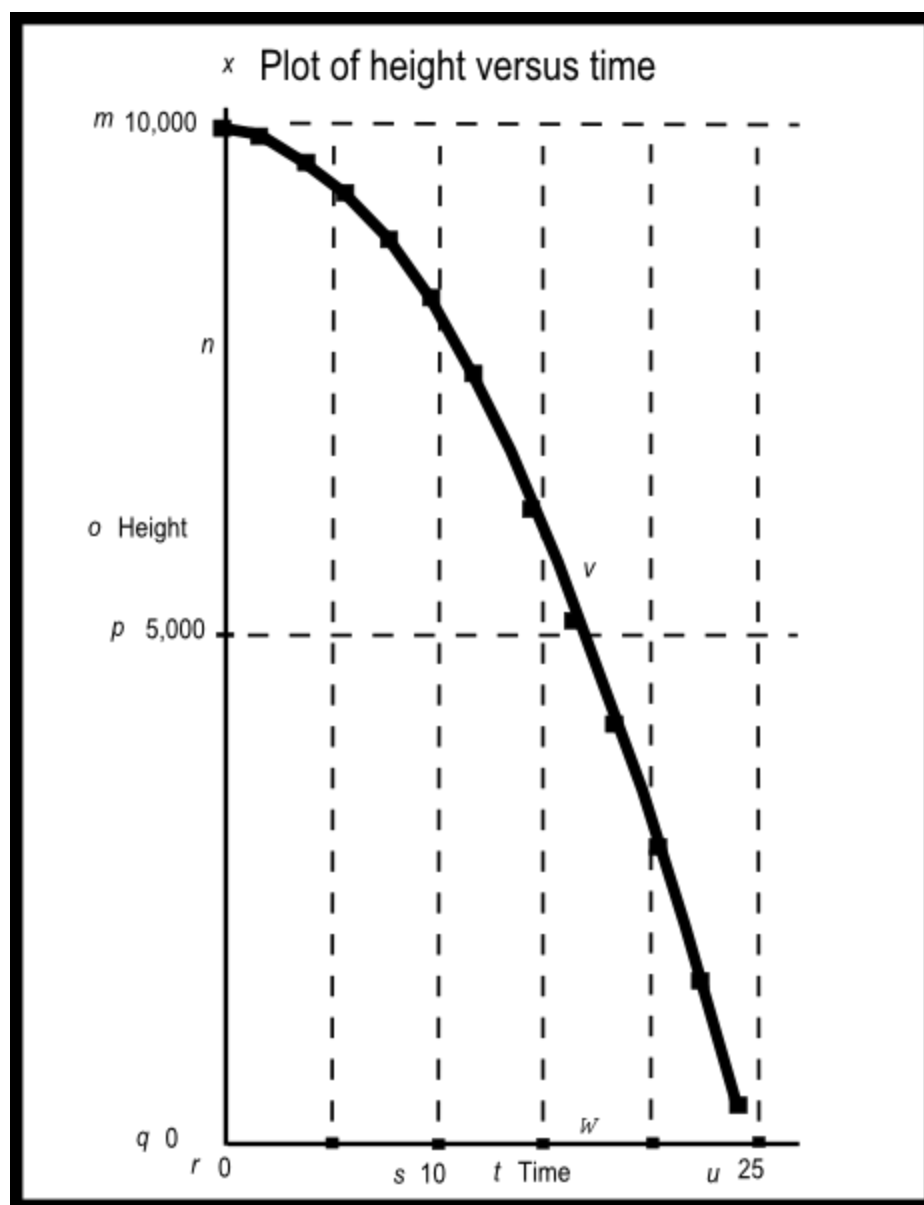
It is stopped by a third player after traveling 20 feet in the northwest direction.

- What is the time required for the puck to complete each leg of its journey?
- What is the average velocity of the puck from the start to the end of its journey?

Vector diagram for the hockey puck

[Figure 2](#) shows a vector diagram for the hockey puck.

Figure 2 - Vector diagram for the hockey puck.



Let's write a script to solve the problem

Please copy the code from [Listing 2](#) into an html file and open it in your browser.

Listing 2 . An exercise on average velocity.

```
<!------- File JavaScript02.html ----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/>");

//The purpose of this function is to receive the
adjacent
// and opposite side values for a right triangle
and to
// return the angle in degrees in the correct
quadrant.
function getAngle(x,y){
    if((x == 0) && (y == 0)){
        //Angle is indeterminate. Just return zero.
        return 0;
    }else if((x == 0) && (y > 0)){
        //Avoid divide by zero denominator.
        return 90;
    }else if((x == 0) && (y < 0)){
        //Avoid divide by zero denominator.
        return -90;
    }else if((x < 0) && (y >= 0)){
        //Correct to second quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else if((x < 0) && (y <= 0)){
        //Correct to third quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else{
```

```

        //First and fourth quadrants. No correction
        required.
        return Math.atan(y/x)*180/Math.PI;
    }//end else
}//end function getAngle

//Compute the time required to traverse each leg.
Identify
// the two legs as A and B.
var vecAmag = 15;//displacement in feet
var vecAang = 45;//displacement angle in degrees
var vecAvel = 50;//velocity magnitude in feet per
second
var timeA = vecAmag/vecAvel;//feet/(feet/second) =
seconds

var vecBmag = 20;//displacement in feet
var vecBang = 135;//displacement angle in degrees
var vecBvel = 60;//velocity magnitude in feet per
second
var timeB = vecBmag/vecBvel;//feet/(feet/second) =
seconds

//Compute the overall displacement
//Compute the horizontal and vertical components
// of each vector.
var vecAh = vecAmag*Math.cos(vecAang*Math.PI/180);
var vecAv = vecAmag*Math.sin(vecAang*Math.PI/180);

var vecBh = vecBmag*Math.cos(vecBang*Math.PI/180);
var vecBv = vecBmag*Math.sin(vecBang*Math.PI/180);

//Compute the sums of the horizontal and vertical
// components from the two vectors to get the
// horizontal and vertical component of the
// resultant vector.
var vecResultH = vecAh + vecBh;

```

```

var vecResultV = vecAv + vecBv;

//Use the Pythagorean theorem to compute the
magnitude of the
// resultant vector in feet.
var vecResultMag =
Math.sqrt(Math.pow(vecResultH,2) +
           Math.pow(vecResultV,2));

//Compute the angle of the resultant vector in
degrees.
vecResultAng = getAngle(vecResultH,vecResultV);

var totalTime = timeA + timeB;//seconds
var vecVelMag =
vecResultMag/totalTime;//feet/second
var vecVelAng = vecResultAng;//degrees


document.write("Time for leg A = " +
timeA.toFixed(2) +
           " seconds<br/>");
document.write("Time for leg B = " +
timeB.toFixed(2) +
           " seconds<br/>");
document.write("Displacement magnitude = " +
           vecResultMag.toFixed(2) + "
feet<br/>");
document.write("Displacement angle = " +
           vecResultAng.toFixed(2) + "
degrees<br/>");
document.write("Total time = " +
totalTime.toFixed(2) +
           " seconds<br/>");
document.write("Average velocity magnitude = " +
           vecVelMag.toFixed(2) + "
feet/second<br/>");

```



```
document.write("Average velocity angle = " +  
                vecVelAng.toFixed(2) + "  
degrees<br/>");  
  
document.write("End Script");  
  
</script>  
</body></html>
```

Screen output

The text shown in [Figure 3](#) should appear in your browser window when the html file is opened in your browser.

Figure 3 . Screen output for Listing #2.

```
Start Script  
Time for leg A = 0.30 seconds  
Time for leg B = 0.33 seconds  
Displacement magnitude = 25.00 feet  
Displacement angle = 98.13 degrees  
Total time = 0.63 seconds  
Average velocity magnitude = 39.47 feet/second  
Average velocity angle = 98.13 degrees  
End Script
```

Analysis of the code

In this exercise, the puck makes a journey across the ice consisting of two sequential legs in different directions with different magnitudes of velocity.

Although each leg involves motion along a line, the total trip is not along a line.

The objective of the script is to

- Compute and display the time required to traverse each leg of the trip based on the given information.
- Compute and display the magnitude and the angle of the displacement from start to finish.
- Compute and display the time required to complete the entire trip, which is the sum of the times from both legs.
- Compute and display the magnitude of the average velocity as the magnitude of the displacement divided by the total time.
- Recognize that the angle of the average velocity is the same as the angle of the displacement.

The **getAngle** function

The code in [Listing 2](#) begins with the **getAngle** function that we developed and used an earlier module. The purpose of the function is to receive the adjacent and opposite side values for a right triangle as parameters and to return the angle in degrees in the correct quadrant.

I explained this function in an earlier module and won't repeat that explanation in this module.

Compute the time for each leg

Following the definition of the **getAngle** function, [Listing 2](#) computes the time required to traverse each leg of the trip. For variable naming purposes, the two legs are identified as A and B.

Variable names that end with "mag" contain vector magnitudes. Variable names that end with "ang" contain vector angles. Variable names that end with "vel" contain the magnitudes of straight-line, uniform velocities.

The time to complete each leg of the trip is computed by dividing the magnitude of the displacement vector for that leg by the magnitude of the

straight-line velocity for that leg. The resulting times have units of seconds, and are saved in the variables named **timeA** and **timeB** .

The overall displacement

The magnitude and angle of the overall displacement for the two-leg trip (identified by the variables named **vecResultMag** and **vecResultAng** respectively) are computed using procedures that were explained in an earlier module. Therefore, I won't repeat that explanation in this module.

The magnitude of the overall displacement is stored in the variable named **vecResultMag** , and the angle for the overall displacement is stored in the variable named **vecResultAng** .

The total time

The total time for the trip, identified by the variable named **totalTime** , is computed as the sum of the times for the individual legs.

Magnitude of the average velocity vector

The magnitude of the average velocity vector, identified by the variable named **vecVelMag** , is compute by dividing the magnitude of the displacement vector by the total time. This results in a value having units of feet/second as shown by the comments.

The direction of the average velocity vector

The direction of the average velocity vector, identified by the variable named **vecVelAng** , is recognized as being the same as the direction of the overall displacement vector with units of degrees.

Display the results

Finally, the **document.write** method is called several times in succession to display the output text shown in [Figure 3](#) .

Multiple concurrent velocities

When a body has multiple concurrent velocities, the overall velocity of the body is equal to the vector sum of the individual velocities. You can compute that vector sum in any way that works for you, including the parallelogram rule, a tail-to-head vector diagram, or the mathematical techniques that we will use here.

Relative velocity

In this section, we will also be dealing with a topic called **relative velocity** . For example, suppose we see a man walking down the aisle in a passenger car of a train that is moving slowly at a uniform velocity along a straight track. How fast is the man moving?

The frame of reference

The answer to that question depends on the frame of reference of the observer. For example, to another passenger in the same rail car, it may appear that the man is moving at about 3 feet per second, which is a reasonably comfortable walking speed.

However, to someone standing on the ground outside of the passenger car (pretend that the side of the car is transparent), it may appear that the man is moving much faster or much slower than 3 feet per second, depending on which direction the man is moving relative to the motion of the train.

It could even turn out that insofar as the outside observer is concerned, the man isn't moving at all, or is moving backwards.

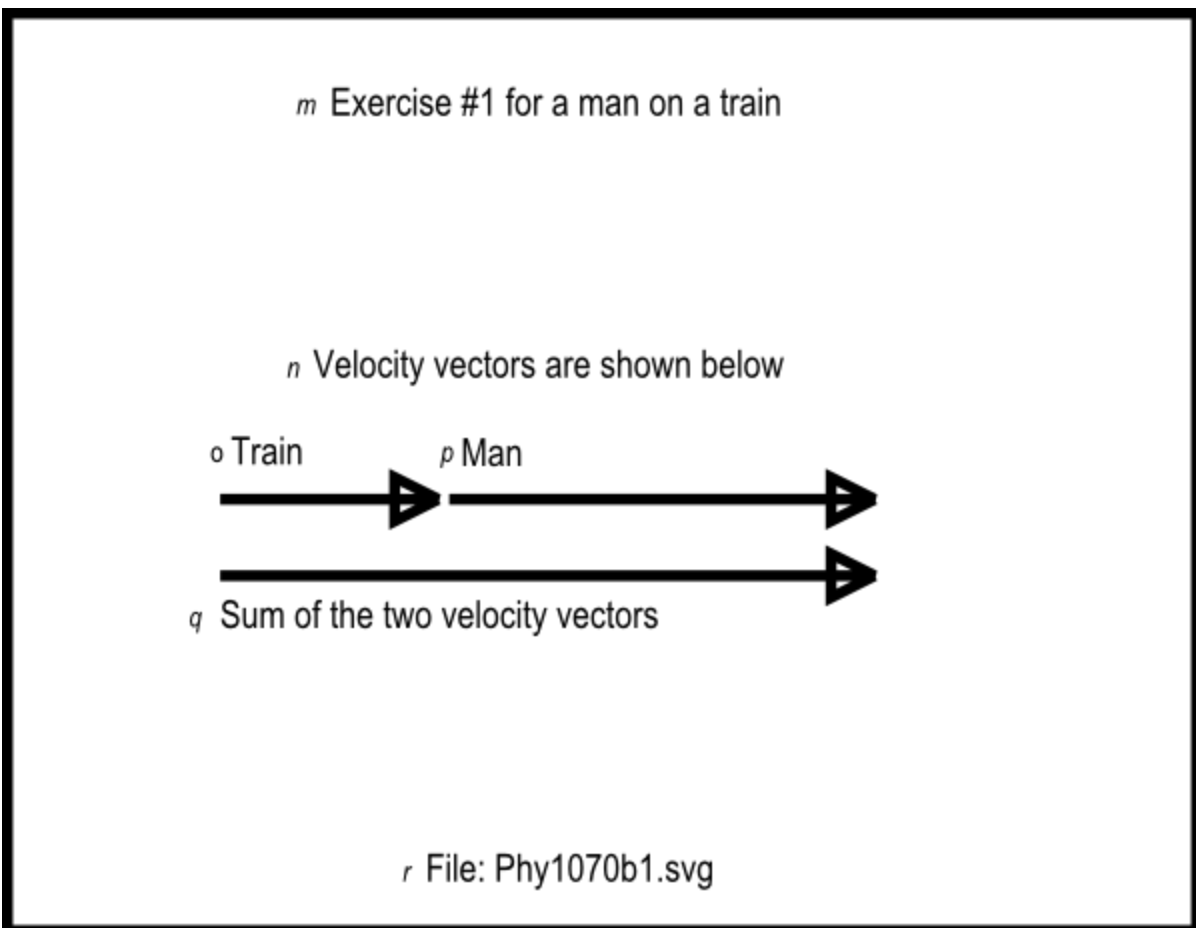
Exercise #1 for a man on a train

A man is walking along the aisle in a passenger car of a train that is moving at 1 mile per hour toward the east. The man is walking in the same direction that the train is moving. The man is walking at a uniform velocity of 2.933 feet per second. (You will see shortly why I chose such a strange walking

speed for the man.) What is the man's overall velocity with reference to the ground?

[Figure 4](#) shows a vector diagram for the man on the train. The units for both velocity vectors must be the same. Therefore, the length of the velocity vector for the man is based on a conversion from 2.933 feet per second to 2 miles per hour.

Figure 4 - Vector diagram for a man on a train.



JavaScript code

Please copy the code shown in [Listing 3](#) into an html file and open the file in your browser.

Listing 3 . Exercise #1 for a man on a train.

```

<!------- File JavaScript03.html -----
----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/>");

//The purpose of this function is to receive the
adjacent
// and opposite side values for a right triangle
and to
// return the angle in degrees in the correct
quadrant.
function getAngle(x,y){

    if((x == 0) && (y == 0)){
        //Angle is indeterminate. Just return zero.
        return 0;
    }else if((x == 0) && (y > 0)){
        //Avoid divide by zero denominator.
        return 90;
    }else if((x == 0) && (y < 0)){
        //Avoid divide by zero denominator.
        return -90;
    }else if((x < 0) && (y >= 0)){
        //Correct to second quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else if((x < 0) && (y <= 0)){
        //Correct to third quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else{
        //First and fourth quadrants. No correction
        required.
        return Math.atan(y/x)*180/Math.PI;
    } //end else
} //end function getAngle
//-----

```

```
-----//
```

```
//The purpose of this function is to add two
vectors, given
// the magnitude and angle (in degrees) for each
vector.
// The magnitude and angle (in degrees) of the
resultant
// vector is returned in a two-element array, with
the
// magnitude in the element at index 0 and the
angle in the
// element at index 1. Note that this function
calls the
// getAngle function, which must also be provided
in the
// script. To use this function to subtract one
vector from
// another, add 180 degrees to the angle for the
subtrahend
// vector before passing the angle to the
function. To use
// this function to add more than two vectors, add
the first
// two vectors as normal, then add the output from
this
// function to the third vector, etc., until all
of the
// vectors have been included in the sum.
function
vectorSum(vecAmag,vecAang,vecBmag,vecBang){
    var vecResult = new Array(2);
    //Compute the horizontal and vertical components
    // of each vector.
    var vecAh =
vecAmag*Math.cos(vecAang*Math.PI/180);
    var vecAv =
```

```

vecAmag*Math.sin(vecAang*Math.PI/180);

    var vecBh =
vecBmag*Math.cos(vecBang*Math.PI/180);
    var vecBv =
vecBmag*Math.sin(vecBang*Math.PI/180);

    //Compute the sums of the horizontal and
vertical
    // components from the two vectors to get the
    // horizontal and vertical component of the
    // resultant vector.
    var vecResultH = vecAh + vecBh;
    var vecResultV = vecAv + vecBv;

    //Use the Pythagorean theorem to compute the
magnitude of
    // the resultant vector.
    vecResult[0] = Math.sqrt(Math.pow(vecResultH,2)
+
                                Math.pow(vecResultV,2));

    //Compute the angle of the resultant vector in
degrees.
    vecResult[1] = getAngle(vecResultH,vecResultV);

    return vecResult;
}//end vectorSum function
//-----
-----//

//The main script body begins here.

//Establish the magnitude and angle for each
vector.
var vecTrainMag = 1;//magnitude of velocity of
train in miles/hr

```



```

var vecTrainAng = 0;//angle of velocity of train
in degrees

var vecManMag = 2.933*3600*(1/5280);//Magnitude of
velocity of
                                // man (ft/sec)*
(sec/hr)*(mile/ft)
                                // = miles/hour
var vecManAng = 0;//angle of velocity of man in
degrees

//Add the two vectors
var resultant = vectorSum(vecTrainMag,vecTrainAng,
                           vecManMag,vecManAng);

//Display the magnitude and direction of the
resultant vector.
document.write("Velocity magnitude = " +
               resultant[0].toFixed(2) + "
miles/hour<br/>");
document.write("Velocity angle = " +
               resultant[1].toFixed(2) + "
degrees<br/>");

document.write("End Script");

</script>
</body></html>

```

Screen output

The text shown in [Figure 5](#) should appear in your browser window when you open the html file in your browser.

Figure 5 . Screen output for Listing #3.

```
Start Script
Velocity magnitude = 3.00 miles/hour
Velocity angle = 0.00 degrees
End Script
```

Analysis of the code

As before, the script begins by defining the **getAngle** function, which doesn't require further explanation.

The vectorSum function

As you can see from [Listing 3](#), this script contains a new function named **vectorSum** . The purpose of this function is to add two vectors, given the magnitude and angle (in degrees) for each vector as incoming parameters. (The function assumes that both incoming magnitude parameters are expressed in the same units.)

The need to add two vectors occurs so often in physics that I decided to encapsulate the capability into a function that we can copy into future scripts. That will relieve us of the need to rewrite the code each time we need that functionality.

Return an array object

The magnitude and the angle (in degrees) of the resultant vector is returned in a two-element array object, with the magnitude in the element at index 0 and the angle in the element at index 1. I will have a more to say about arrays shortly.

Note that this function calls the **getAngle** function, which must also be provided in the script.

Subtraction of vectors

The function can also be used to subtract one vector from another. To subtract one vector from another, simply add 180 degrees to the angle for the subtrahend vector before passing the angle to the function.

Adding more than two vectors

Although the function can only be used to add two vectors, it can be called repeatedly to add any number of vectors two at a time.

To use this function to add more than two vectors, add the first two vectors as normal. Then add the vector output from the function to the third vector, etc., until all of the vectors have been included in the sum.

The code in the vectorSum function

Most of the code in the function is the same as code that I have explained in earlier modules. However, there are a couple of lines of code that are different.

The function begins with the following statement:

```
var vecResult = new Array(2);
```

The purpose of this statement is to create a two-element array object named **vecResult** .

What is an array object?

You can think of an array object as a software object containing pigeon holes into which you can store data and from which you can later retrieve that data. This particular array object has two pigeon holes, and they are numbered 0 and 1.

This array object will be used as a container to return the magnitude and angle values for the resultant vector.

Can only return one item

A JavaScript function can only return one item. Up to this point in these modules, that item has consisted of a single numeric value, such as the angle value that is returned from the **getAngle** function.

However, the **vectorSum** function needs to return two different values. One way to do that is to put those two values in the pigeon holes of an array object and return that array object as the one item that can be returned.

Store the resulting magnitude in the array object

Once the array object is created, the code in the function shown in [Listing 3](#) is essentially the same as the code used earlier to add two vectors, down to the statement that begins with:

```
vecResult[0] = Math.sqrt(Math.pow(vecResultH,2) +...
```

That statement computes the magnitude of the resultant vector the same way as before, but instead of storing the value of the magnitude in a variable, it is stored in the first pigeon hole of the array object.

The value is directed into the first pigeon hole (technically called element) by the "[0]" that you see following the name of the array object in that statement.

Store the resulting angle in the array object

Immediately thereafter, the statement in [Listing 3](#) that reads

```
vecResult[1] = getAngle(vecResultH,vecResultV);
```

computes the angle for the resultant vector and stores it in the second element in the array object (the element identified by the index value 1).

Return the array object for use later

These same index values along with the square brackets "[]" will be used later to retrieve the magnitude and angle values from the array object.

Finally, the function returns the array object to the calling script by executing the statement that reads

```
return vecResult;
```

The main script body

The main script body begins where indicated by the comment in [Listing 3](#).

The first four statements establish values for the magnitude and direction of the two vectors that represent the train and the man. There is nothing in those statements that you haven't seen before.

Note the treatment of the units

However, I encourage you to pay attention to the treatment of units in the comments for the value stored in the variable named **vecManMag** . The arithmetic in that statement uses two different conversion factors to convert the value from feet per second to miles per hour.

Once again, working through the units in this fashion can help you to organize your arithmetic correctly.

Call the vectorSum function

The statement that begins

```
var resultant = vectorSum(vecTrainMag...
```

calls the new **vectorSum** function to add the two vectors, passing the magnitude and angle for each vector as parameters.

Store the returned array object in a variable

This statement also declares a new variable named **resultant** . The array object that is returned from the **vectorSum** function is stored in this variable.

Once this statement finishes executing, the magnitude and angle of the resultant vector have been computed and saved for later use. (In this script, the only use of those two values is to display them later. However, they will be used in a more significant way in another exercise later.)

Code to display the results

The call to the **vectorSum** function is followed by three calls to the **document.write** method. The first two calls display magnitude and angle values, and the third call simply displays some text to indicate that the script has finished executing.

Displaying the magnitude value

If you examine the statement containing the first call to the **document.write** method, you will see that the argument list contains the following expression:

```
resultant[0].toFixed(2)
```

In this case, the "[0]" means to retrieve the value stored in the array element identified by the index value 0.

Once that value has been retrieved, the built-in **toFixed** method is called, passing the literal value 2 as a parameter, to format the value so that it will be displayed with two digits to the right of the decimal point.

[Figure 5](#) shows that value displayed in the line of text that reads:

Velocity magnitude = 3.00 miles/hour

(Note that very few of the values that I display in these modules comply with the rules for decimal digits and significant figures that I explained in an earlier module.)

A partial solution to the problem

A partial answer to the question posed for this exercise is that the magnitude of the man's overall velocity with respect to the ground is 3

miles per hour as shown in [Figure 5](#).

The rest of the solution

The next statement in [Listing 3](#) uses a similar syntax to retrieve the value from the second element in the array object (the element with an index value of 1) and to display it as

Velocity angle = 0.00 degrees

as shown in [Figure 5](#).

This means that the man is moving due east (the same direction that the train is moving) with a velocity of 3 miles per hour with reference to the ground.

Analysis of the results

Now I will tell you why I used a strange walking speed (2.933 feet per second) for the man. As it turns out, this is very close to 2 miles per hour, and I wanted the man to have a walking speed that would result in simple numeric results.

Let's review the problem

The train is moving east with a uniform velocity of 1 mile per hour relative to the ground.

The man is moving east with a uniform velocity of 2 miles per hour (2.933 feet per second) relative to the train.

What is the man's velocity relative to the ground?

The algebraic sum of the magnitudes

Because the man and the train are both moving along the same straight line, and the man's velocity is stated relative to the train, the magnitude of the man's velocity relative to the ground is the algebraic sum of the magnitudes

of the two velocities. Because they are both moving in the same direction, that algebraic sum is additive.

An analysis of the velocities

The train is moving 1 mile per hour with reference to the ground, and the man is moving 2 miles per hour with reference to the train (along the same line and in the same direction as the train). Therefore, the man is moving $1+2=3$ miles per hour with reference to the ground.

This means that an observer standing on the ground at a point on a line perpendicular to the train would perceive the man to be moving to the right with a velocity of 3 miles per hour (assuming that the train is moving to the right).

A zero-degree angle is not required

Note that it isn't necessary that the train and the man be moving at an angle of zero degrees (east). The magnitude of the result would be the same regardless of the direction that they are moving provided they are moving along the same straight line.

We will specify a different common direction for the train and the man in the next exercise.

Exercise #2 for a man on a train

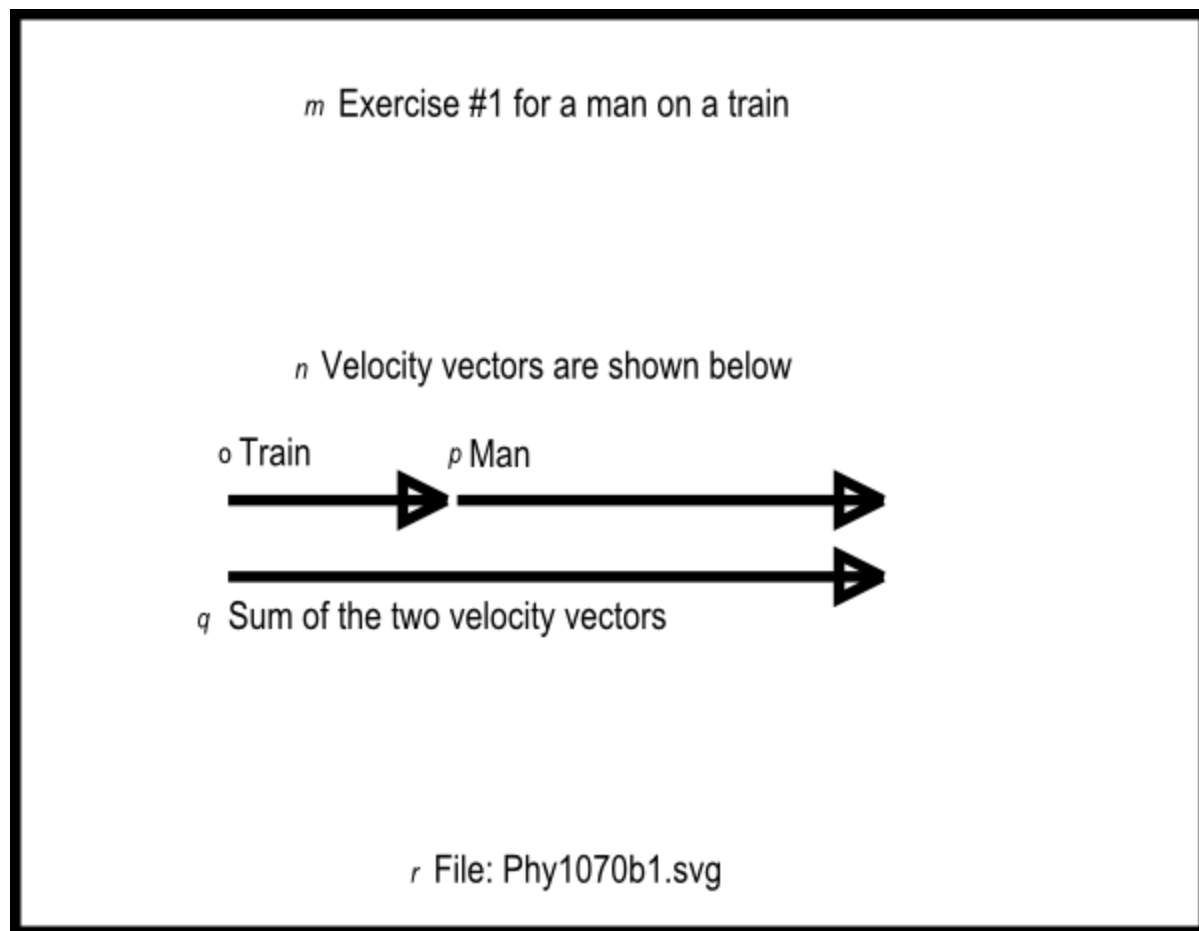
A man is walking in a passenger car of a train that is moving at 1 mile per hour toward the northeast (45 degrees). The man is walking in the opposite direction than the train is moving at 2.933 feet per second. (This means that the angle for the man's velocity vector is 225 degrees.)

What is the man's velocity with reference to the ground?

Another vector diagram for a man on a train

[Figure 6](#) shows another vector diagram for a man on a train. Once again, the units for both velocity vectors must be the same. Therefore, the length of the velocity vector for the man is based on a conversion from 2.933 feet per second to 2 miles per hour.

Figure 6 - Another vector diagram for a man on a train.



Make a new html file

Please copy the code from the previous exercise into a new html file, make the changes described below, and open the html file in your browser.

- Change the value of vecTrainAng from 0 to 45.
- Change the value of vecManAng from 0 to 225.

(Because of the small differences between this script and the previous script, I won't publish a copy of this new script here.)

Screen output

The text shown in [Figure 7](#) should appear in your browser window when you open the file in your browser.

Figure 7 . Screen output for Exercise #2 for a man on a train.

```
Start Script
Velocity magnitude = 1.00 miles/hour
Velocity angle = 225.00 degrees
End Script
```

Analysis of the results

The man and the train are still moving along the same straight line even though the angle is no longer 0. Therefore, the magnitudes of the two vectors still add algebraically. In this case, however, the man and the train are moving in opposite directions, so they are no longer additive.

The man has a greater velocity magnitude

The magnitude of the man's velocity in the direction of 225 degrees is greater than the magnitude of the train's velocity in the opposite direction of 45 degrees.

As you can see from [Figure 7](#), the man's velocity with reference to the ground is 1 mile per hour at an angle of 225 degrees. This means that an observer standing on the ground at a point on a line perpendicular to the train would perceive the man to be moving to the left at 1 mile per hour (assuming that the train is moving to the right).

An exercise with three vectors in a plane

Now we are going to take a look at an exercise involving three vectors in a plane, which are not in a line.

An aircraft carrier is steaming east with a uniform velocity of 2 miles per hour relative to the ground beneath the ocean. A large platform on the deck is sliding northeast with a uniform velocity of 2 miles per hour relative to the ship. A man is walking north on the platform with a uniform velocity of 2 miles per hour relative to the platform.

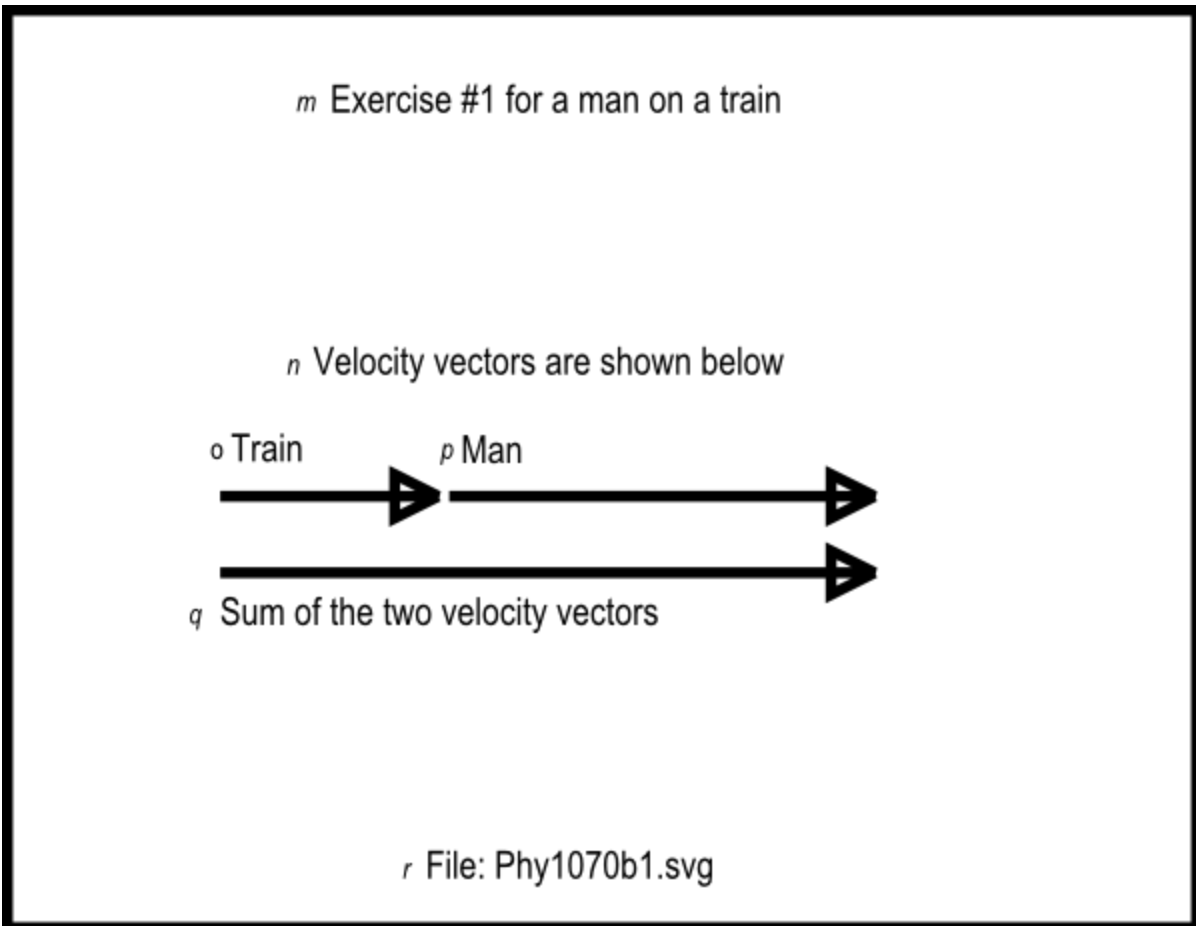
What is the velocity of the platform with reference to the ground below the ocean?

What is the velocity of the man with reference to the ground below the ocean?

Vector diagram for man on aircraft carrier

[Figure 8](#) shows a vector diagram for the man on the aircraft carrier.

Figure 8 - Vector diagram for man on aircraft carrier.



Create the script

Please copy the code from [Listing 4](#) into an html file and open the html file in your browser.

Listing 4 . An exercise with three vectors in a plane.

```
<!------- File JavaScript04.html -----
----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/>");

//The purpose of this function is to receive the
```

```

adjacent
// and opposite side values for a right triangle
and to
// return the angle in degrees in the correct
quadrant.
function getAngle(x,y){

    if((x == 0) && (y == 0)){
        //Angle is indeterminate. Just return zero.
        return 0;
    }else if((x == 0) && (y > 0)){
        //Avoid divide by zero denominator.
        return 90;
    }else if((x == 0) && (y < 0)){
        //Avoid divide by zero denominator.
        return -90;
    }else if((x < 0) && (y >= 0)){
        //Correct to second quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else if((x < 0) && (y <= 0)){
        //Correct to third quadrant
        return Math.atan(y/x)*180/Math.PI + 180;
    }else{
        //First and fourth quadrants. No correction
required.
        return Math.atan(y/x)*180/Math.PI;
    }//end else
}//end function getAngle
//-----
-----//

//The purpose of this function is to add two
vectors, given
// the magnitude and angle (in degrees) for each
vector.
// The magnitude and angle (in degrees) of the
resultant

```

```

// vector is returned in a two-element array, with
the
// magnitude in the element at index 0 and the
angle in the
// element at index 1. Note that this function
calls the
// getAngle function, which must also be provided
in the
// script. To use this function to subtract one
vector from
// another, add 180 degrees to the angle for the
subtrahend
// vector before passing the angle to the
function. To use
// this function to add more than two vectors, add
the first
// two vectors as normal, then add the output from
this
// function to the third vector, etc., until all
of the
// vectors have been included in the sum.
function
vectorSum(vecAmag,vecAang,vecBmag,vecBang){
    var vecResult = new Array(2);
    //Compute the horizontal and vertical components
    // of each vector.
    var vecAh =
vecAmag*Math.cos(vecAang*Math.PI/180);
    var vecAv =
vecAmag*Math.sin(vecAang*Math.PI/180);

    var vecBh =
vecBmag*Math.cos(vecBang*Math.PI/180);
    var vecBv =
vecBmag*Math.sin(vecBang*Math.PI/180);

    //Compute the sums of the horizontal and

```

```

vertical
    // components from the two vectors to get the
    // horizontal and vertical component of the
    // resultant vector.
    var vecResultH = vecAh + vecBh;
    var vecResultV = vecAv + vecBv;

    //Use the Pythagorean theorem to compute the
magnitude of
    // the resultant vector.
    vecResult[0] = Math.sqrt(Math.pow(vecResultH,2)
+
                                Math.pow(vecResultV,2));

    //Compute the angle of the resultant vector in
degrees.
    vecResult[1] = getAngle(vecResultH,vecResultV);

    return vecResult;
} //end vectorSum function
//-----
-----//

//Main body of script begins here.

//Establish the magnitude and angle for each
vector.
var vecShipMag = 2;//magnitude of velocity of ship
in miles/hr
var vecShipAng = 0;//angle of velocity of ship in
degrees

var vecPlatMag = 2;//magnitude of velocity of
platform miles/hr
var vecPlatAng = 45;//angle of velocity of
platform in degrees

```

```
var vecManMag = 2;//Magnitude of velocity of man
in miles/hr
var vecManAng = 90;//angle of velocity of man in
degrees

//Add two vectors
var platformVel =

vectorSum(vecShipMag,vecShipAng,vecPlatMag,vecPlat
Ang);
//Add in the third vector
var manVel =

vectorSum(platformVel[0],platformVel[1],vecManMag,
vecManAng);

//Display the magnitude and direction of the
resultant vectors.
document.write("Platform velocity magnitude = " +
               platformVel[0].toFixed(2) + "
miles/hour<br/>");
document.write("Platform velocity angle = " +
               platformVel[1].toFixed(2) + "
degrees<br/>");
document.write("Man velocity magnitude = " +
               manVel[0].toFixed(2) + "
miles/hour<br/>");
document.write("Man velocity angle = " +
               manVel[1].toFixed(2) + "
degrees<br/>");

document.write("End Script");

</script>
</body></html>
```


Screen output

The text shown in [Figure 9](#) should appear in your browser window when the html file is opened in your browser.

Figure 9 . Screen output for Listing #4.

```
Start Script
Platform velocity magnitude = 3.70 miles/hour
Platform velocity angle = 22.50 degrees
Man velocity magnitude = 4.83 miles/hour
Man velocity angle = 45.00 degrees
End Script
```

Analysis of the code

The velocity of the platform relative to the ground is the vector sum of the ship's velocity and the velocity of the platform relative to the ship. The man's velocity is the vector sum of the platform's velocity relative to the ground and the man's velocity relative to the platform.

[Listing 4](#) begins with copies of the **getAngle** function and the **vectorSum** function, which I have already explained. This discussion begins at the comment that reads "Main body of script begins here."

The main body of the script

The main body begins with the declaration and initialization of six variables whose contents represent the magnitude and the angle of the ship, the platform, and the man.

Add the first two vectors

The **vectorSum** function is called to add the ship's velocity vector (relative to the ground) and the platform's velocity vector (relative to the ship). The resultant vector produced by adding those two vectors is stored in the array-object variable named **platformVel** for use later. This is the platform's velocity vector relative to the ground.

Add the man's vector to the sum

Then the **vectorSum** function is called again to add the platform's velocity vector (relative to the ground) and the man's velocity vector (relative to the platform). Note that the two values stored in the **platformVel** array object are extracted and passed as parameters to the **vectorSum** function.

The resultant vector produced by that addition is saved in the array-object variable named **manVel**. This is the man's velocity vector relative to the ground.

Display the results

Finally, the method named **document.write** is called four times in succession to extract and print the four values stored in the two array objects, producing the text output shown in [Figure 9](#).

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of `.html`. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302-0370 Motion -- Uniform and Relative Velocity
- File: Game0370.htm
- Published: 10/13/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

GAME 2302-0380 Motion -- Variable Velocity and Acceleration
The purpose of this module is to explain variable velocity and acceleration.

Table of Contents

- [Preface](#)
 - [General](#)
 - [Viewing tip](#)
 - [Figures](#)
 - [Listings](#)
- [General background information](#)
 - [Variable velocity](#)
 - [Acceleration](#)
 - [The acceleration of gravity](#)
- [Discussion and sample code](#)
 - [Variable velocity exercise #1](#)
 - [Variable velocity exercise #2](#)
 - [Acceleration of gravity exercise #1](#)
 - [Acceleration of gravity exercise #2](#)
 - [Acceleration of gravity exercise #3](#)
 - [Other useful equations](#)
 - [Exercise to find the velocity](#)
 - [Exercise to find the height](#)
- [Run the scripts](#)
- [Miscellaneous](#)

Preface

General

This module is part of a series of modules designed for teaching the physics component of *GAME2302 Mathematical Applications for Game Development* at Austin Community College in Austin, TX. (See [GAME 2302-0100: Introduction](#) for the first module in the course along with a description of the course, course resources, homework assignments, etc.)

The purpose of this module is to explain variable velocity and acceleration.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

Figures

- [Figure 1](#). Graph for variable velocity exercise #1.
- [Figure 2](#). Screen output for Listing #1.
- [Figure 3](#). Displacement versus time for first five time intervals.
- [Figure 4](#). Screen output for Listing #2.
- [Figure 5](#). Screen output for Listing #3.
- [Figure 6](#). Screen output for Listing #4 at 45 degrees.
- [Figure 7](#). Trajectory for two different firing angles.
- [Figure 8](#). Screen output for Listing #4 at 60 degrees.
- [Figure 9](#). Screen output for Listing #5.
- [Figure 10](#). Screen output for Listing #6.
- [Figure 11](#). Screen output for Listing #7.

Listings

- [Listing 1](#). Variable velocity exercise #1.
- [Listing 2](#). Variable velocity exercise #2.
- [Listing 3](#). Acceleration of gravity exercise #1.

- [Listing 4](#). Acceleration of gravity exercise #2.
- [Listing 5](#). Acceleration of gravity exercise #3.
- [Listing 6](#). Exercise to find the velocity.
- [Listing 7](#). Exercise to find the height.

General background information

I will provide an introduction to variable velocity, acceleration, and the acceleration of gravity in this section. I will provide exercises on those topics in the next section.

Variable velocity

What is velocity?

To review, you learned in an earlier module that velocity is the rate of change of position. Since displacement is a change in position, velocity is also the rate of displacement.

Uniform versus variable velocity

Previous modules have dealt exclusively with uniform velocity. However, many situations in life involve variable velocity and acceleration. For example, the simple act of stopping an automobile at a traffic light and then resuming the trip once the light turns green involves variable velocity and acceleration.

Three equations

I will explain both of these topics in this module using three well-known physics equations that deal with the displacement of a projectile in a vacuum **under the influence of gravity**.

The equation that we will spend the most time on is

$$h = v_0 * t + 0.5 * g * t^2$$

where

- h is the distance of the projectile above the surface of the earth
- v_0 is the initial velocity of the projectile
- t is time in seconds
- g is the acceleration of gravity, approximately 9.8 meters per second squared, or approximately 32.2 feet per second squared at the surface of the earth. (We will also do an exercise involving the acceleration of gravity on the moon.)

(I will provide the other two equations [later](#).)

Acceleration

Everyone is familiar with the acceleration that occurs when a motor vehicle speeds up or slows down. When the vehicle speeds up very rapidly, the positive acceleration forces us against the back of the seat. (This involves the relationship among force, mass, and acceleration, which will be the subject of a future module.)

If the vehicle slows down very rapidly or stops suddenly, the negative acceleration may cause us to crash into the windshield, the dashboard, or a deployed airbag.

The accelerator pedal

A common name for the pedal that causes gasoline to be fed to the engine is often called the accelerator pedal because it causes the vehicle to speed up. (However, I have never heard anyone refer to the pedal that causes the vehicle to slow down as the deceleration pedal. Instead, it is commonly called the brake pedal.)

Definitions

Displacement is a change in position.

Velocity is the rate of change of position or the rate of *displacement* .

Acceleration is the rate of change of *velocity* .

Jerk is the rate of change of *acceleration* (not covered in this module).

According to [this author](#) , there is no universally accepted name for the rate of change of *jerk* .

The algebraic sign of acceleration

When the velocity of a moving object increases, that is viewed as positive acceleration. When the velocity of the object decreases, that is viewed as negative acceleration.

Uniform or variable acceleration

Acceleration may be uniform or variable. It is uniform only if equal changes in velocity occur in equal intervals of time.

A vector quantity

Acceleration has both direction and magnitude. Therefore, acceleration is a vector quantity.

The units for acceleration

The above [definition](#) for acceleration leads to some interesting units for acceleration. For example, consider a situation in which the velocity of an object changes by 5 feet/second in a one-second time interval. Writing this as an algebraic expression gives us

$(5 \text{ feet/second})/\text{second}$

Multiplying the numerator and the denominator of the fraction by 1/second gives us

$5 \text{ feet}/(\text{second}*\text{second})$

This is often written as

5 feet/second²

which is pronounced five feet per second squared.

The acceleration of gravity

The exercises in the remainder of this module are based on the following two assumptions:

- For practical purposes, the effect of the acceleration of gravity is the same regardless of the height of an object above the surface of the earth, provided that the distance above the surface of the earth is small relative to the radius of the earth.
- In the absence of an atmosphere, all objects fall toward the earth with the same acceleration regardless of their masses.

Let's examine the validity of these two assumptions.

Newton's law of universal gravitation

This [law](#) states that every point mass in the universe attracts every other point mass with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between them. (Separately it has been shown that large spherically symmetrical masses attract and are attracted as if all their mass were concentrated at their centers.)

Effect of gravity versus altitude

An object at the surface of the earth is approximately 3963 miles from the center of the earth. An object that is six miles above the surface of the earth (typical flying altitude for a passenger plane) is approximately 3969 miles from the center of the earth.

Therefore, (if I didn't make an arithmetic error) the gravitational attraction between the earth and that object changes by less than 0.5 percent when the

object is transported from the surface of the earth to a position that is six miles above the surface of the earth.

For practical purposes, therefore, we can assume that the acceleration of gravity is constant up to at least 32000 feet above the surface of the earth.

Acceleration is independent of mass

Newton's law, as applied to the gravitational attraction of the earth, expressed in algebraic terms, looks something like **the following** :

$$f = E * m / d^2$$

where

- f is the attractive force between the earth and another object
- E is the mass of the earth
- m is the mass of the other object
- d is the distance between the center of mass of the earth and the center of mass of the other object

Force equals mass times acceleration

It is a well-known principle of physics that an object that is free to move will move and will accelerate when subjected to a force. The acceleration of the object will be proportional to the force and inversely proportional to its mass. In other words,

$$a = f / m$$

where

- a is the acceleration in units such as meters/sec²
- f is force in units such as kilograms*meters/sec² (newtons)
- m is mass in units such as kilograms

By multiplying both sides of the equation by m, we get a more **common presentation** of this relationship, which is

$$f = m \cdot a$$

where the symbols mean the same as listed [above](#).

A ratio of two different forces of gravity

Now let's use the equation from [above](#) to form a ratio of the forces exerted on two different masses by the earth, assuming that both masses are the same distance from the center of the earth.

$$f_1/f_2 = (E \cdot m_1/d^2)/(E \cdot m_2/d^2)$$

Cancel like terms to simplify

If we cancel like terms from the numerator and denominator of the expression on the right, we can simplify the ratio to

$$f_1/f_2 = m_1/m_2$$

Replacing the forces on the left by the expression from [above](#), we get

$$m_1 \cdot a_1 / m_2 \cdot a_2 = m_1/m_2$$

Multiplying both sides by m_2/m_1 we get

$$a_1/a_2 = 1$$

or

$$a_1 = a_2$$

This shows that the acceleration resulting from the gravitational force exerted on two objects that are equally distant from the center of mass of the earth is the same regardless of the differences in mass of the two objects.

Discussion and sample code

I will present and explain several different scenarios based on the above assumptions in this section.

Variable velocity exercise #1

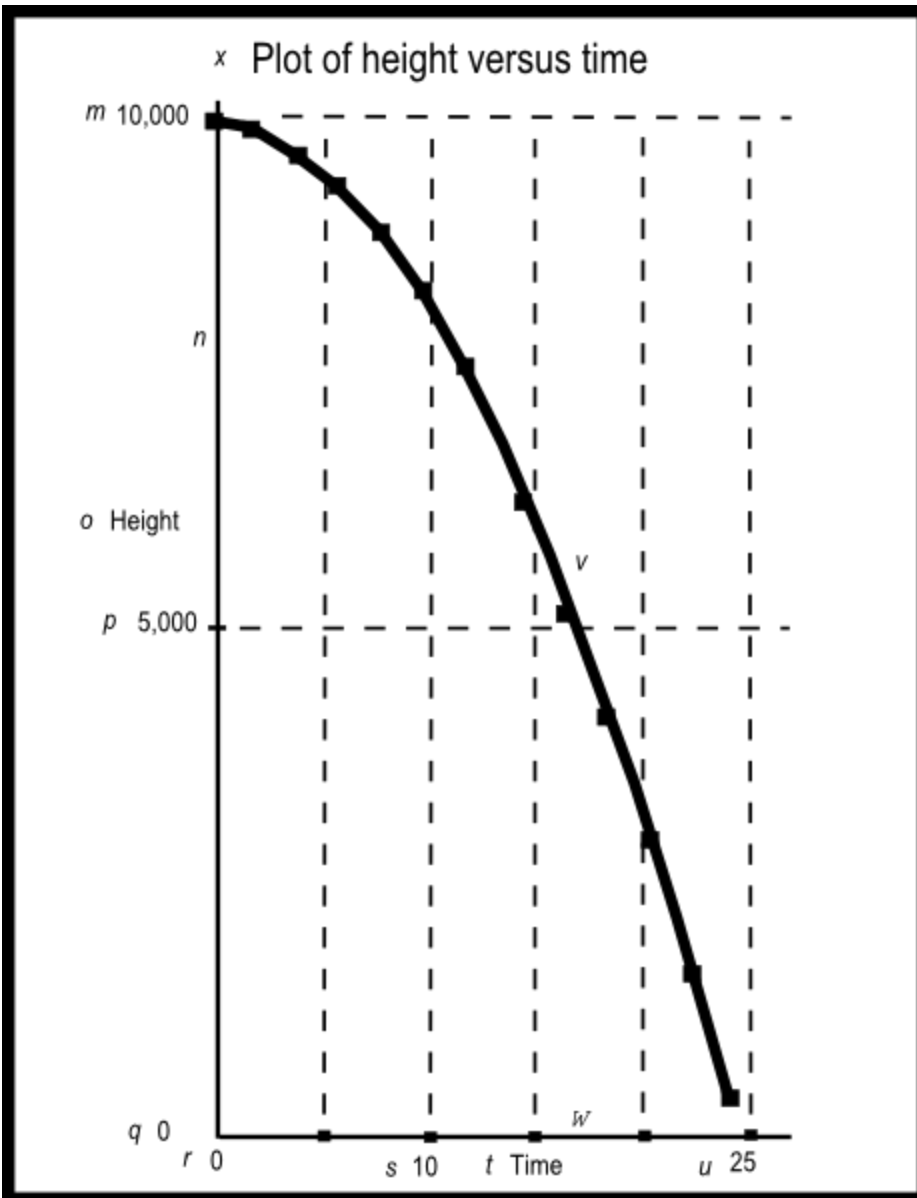
An archer that is six feet tall shoots an arrow directly upward with a velocity of 100 feet per second. Assume the arrow is at a height of 6 feet when it leaves the bow. Also ignore the effects of air resistance.

Plot height versus time

Use your graph paper and plot the height of the arrow on the vertical axis versus time on the horizontal axis. Plot one point each 0.25 seconds. Allow for a maximum time value of about 7 seconds and a maximum height value of about 165 feet.

Your plot should look something like the curve in [Figure 1](#).

Figure 1 - Graph for variable velocity exercise #1.



Create a script

Let's analyze this situation using a script. Copy the code shown in [Listing 1](#) into an html file and open the file in your browser.

Listing 1 . Variable velocity exercise #1.

[illegible]

Listing 1 . Variable velocity exercise #1.

```
h = h0 + v0*t +0.5*g*t*t;

//Display the information for this iteration
document.write(
    "t = " + t.toFixed(2) + " seconds" + sp
+
    " h = " + h.toFixed(1) + " feet" + sp +
    "<br/>");

//Increment the time for the next iteration.
t = t + tInc;
} //end while loop

document.write("End Script");

</script>
</body></html>
```

Screen output

The text shown in [Figure 2](#) should appear in your browser window when you open the html file in your browser.

Figure 2 . Screen output for Listing #1.

Figure 2 . Screen output for Listing #1.

```
Start Script
t = 0.00 seconds      h = 6.0 feet
t = 0.25 seconds      h = 30.0 feet
t = 0.50 seconds      h = 52.0 feet
t = 0.75 seconds      h = 72.0 feet
t = 1.00 seconds      h = 89.9 feet
t = 1.25 seconds      h = 105.8 feet
t = 1.50 seconds      h = 119.8 feet
t = 1.75 seconds      h = 131.7 feet
t = 2.00 seconds      h = 141.6 feet
t = 2.25 seconds      h = 149.5 feet
t = 2.50 seconds      h = 155.4 feet
t = 2.75 seconds      h = 159.2 feet
t = 3.00 seconds      h = 161.1 feet
t = 3.25 seconds      h = 160.9 feet
t = 3.50 seconds      h = 158.8 feet
t = 3.75 seconds      h = 154.6 feet
t = 4.00 seconds      h = 148.4 feet
t = 4.25 seconds      h = 140.2 feet
t = 4.50 seconds      h = 130.0 feet
t = 4.75 seconds      h = 117.7 feet
t = 5.00 seconds      h = 103.5 feet
t = 5.25 seconds      h = 87.2 feet
t = 5.50 seconds      h = 69.0 feet
t = 5.75 seconds      h = 48.7 feet
t = 6.00 seconds      h = 26.4 feet
t = 6.25 seconds      h = 2.1 feet
t = 6.50 seconds      h = -24.2 feet
End Script
```

Analysis of the output

Your curve should begin at a height of 6 feet and the height should increase on a point-by-point basis out to about 3 seconds. The maximum height should occur around 3 seconds and should then start decreasing. The height should go to zero between 6.25 seconds and 6.5 seconds.

Not a trip to outer space

As you are probably aware, shooting an arrow upward does not cause the arrow to go into outer space, unless the arrow is self-propelled and manages to reach a velocity commonly known as the "escape velocity."

Instead, for any practical value of initial velocity (neglecting air resistance), gravitational attraction will eventually cause the arrow to slow down, reverse course, and fall back to earth with continually increasing velocity until it strikes the earth and stops. That is the message conveyed by the plot of height in [Figure 1](#) and the output from the script shown in [Figure 2](#).

The maximum height

At about 3 seconds and a height of about 161 feet, the kinetic energy provided by the initial velocity will have been dissipated by the gravitational attraction of the earth. At that point, the arrow won't go any higher. Instead, it will start falling back toward the earth. Somewhere around 6.25 seconds, it will strike the earth.

The shape of the curve

The shape of this curve is controlled by only two factors: the initial velocity of the arrow and the gravitational attraction of the earth. The archer has some degree of control over the initial velocity, but has no control over the gravitational attraction of the earth.

In theory, in the absence of an atmosphere, if the arrow is shot straight up, the arrow should land in the same place from which it was shot. In practice in the real world, wind and other factors would probably prevent that from happening.

Variable velocity

We learned in an earlier module that if velocity is uniform, the displacement should be the same during each successive equal interval of time. However, we can see from [Figure 2](#) that is not true in this case. For example, [Figure 3](#) shows the displacement versus time for the first five time intervals and we can see that it is anything but uniform.

Figure 3 . Displacement versus time for first five time intervals.

Interval	Displacement
1	24
2	22
3	20
4	17.9
5	15.9

Successively decreasing or increasing displacement

Referring back to [Figure 2](#) (along with [Figure 3](#)), we see that the displacement during the first 0.25 second time interval was 24 feet, but the displacement between 2.75 seconds and 3.0 seconds was only 1.9 feet, Furthermore, the displacement was -0.2 feet during the interval between 3 seconds and 3.25 seconds indicating that the arrow had begun falling back to the earth.

As you can see in [Figure 2](#), from that point until contact with the ground, the displacement increased during each 0.25 second interval.

In this case, the velocity clearly wasn't uniform. Instead it was variable. The velocity decreased as the arrow was going up, and increased as the arrow was falling back down.

Measuring variable velocity in the real world

Measuring variable velocity in the real world can be difficult. What you need to do is to measure the displacement of an object in as short a time interval as possible and then divide the displacement by the value of the time interval. If you can do this at enough points in time, you can construct curves that represent the variable velocity to which the object is being subjected.

Estimating variable velocity graphically

Assuming that you are able to create a plot of displacement versus time at closely-spaced points, you can estimate the variable velocity curve by connecting each pair of points with a straight line and measuring the slope of each such straight line segment. An estimate of the velocity during the interval defined by a pair of points is the slope of the line that connects those points. The closer the points are, the better will be the estimate of the velocity at a point half way between those two points.

In the situation where you are able to collect enough data to draw a smooth curve of displacement versus time, the instantaneous velocity at any point on that curve is the slope of a line that is tangent to the curve at that point. Note, however, that the construction of such a tangent line is difficult.

Units of velocity

If may recall from your high school geometry course that the slope of a line is given by the "rise over run." In other words, the slope of the line is the ratio of the height to the base of a right triangle for which the hypotenuse is on or parallel to the line.

Estimate some variable velocity values

At this point, I encourage you to use an object with a straight edge, place it firmly on two points that define a time interval on your graph, estimate the slope of the edge, and record that slope as your estimate of the velocity at the center of that time interval. We will do something similar in our script.

Explanation of the code

The code in [Listing 1](#) begins by declaring and initializing values to contain the parameters of the problem:

- The acceleration of gravity in feet/sec². Note that this value is negative indicating that the gravitational attraction is toward the center of the earth (down).
- The initial velocity of the arrow in feet/sec. This value is positive because the direction of velocity is away from the center of the earth (up).
- The height of the arrow when it is released, positive values meaning up from the ground.
- The time interval at which successive estimates of the height of the arrow will be computed.

Working variables

Following that, the code in [Listing 1](#) declares and initializes two working variables for time and distance that will change as the program progresses.

Then [Listing 1](#) declares and initializes a variable named **sp** that is used solely to insert spaces between the columns in the output display.

A while loop

A **while** loop is used to evaluate the equation given [earlier](#), to compute and display the height of the arrow every 0.25 seconds for as long as the arrow is above the ground (height is positive). The loop continues to iterate and display time and height values in two separate columns (see [Figure 2](#)) until the computation produces one negative height value.

When the test at the top of the **while** loop determines that the previously-computed value for height was negative,

- the body of the loop is skipped,
- the "End Script" message is displayed, and
- the script terminates.

This is not a complicated script in comparison with some of the scripts that were presented in earlier modules.

Given the output shown in [Figure 2](#), you should be able to estimate and plot the velocity of the arrow on the same graph by dividing the displacement during each time interval by the value of the time interval.

Variable velocity exercise #2

Estimating variable velocity with a computer model

Let's write a script that approximates the graphic procedure that I described above.

Copy the code from [Listing 2](#) into an html file and open the file in your browser.

Listing 2 . Variable velocity exercise #2.

```
<!------- File JavaScript2.html -----  
----->  
<html><body>  
<script language="JavaScript1.3">  
  
document.write("Start Script <br/>");  
  
//Declare and initialize constants  
var g = -32.2;//acceleration of gravity in  
ft/sec/sec  
var v0 = 100;//initial velocity in ft/sec  
var h0 = 6;//initial height in feet  
var tInc = 0.25;//calculation interval in seconds  
  
//Declare and initialize working variables  
var t = .0001;//current time in seconds
```

```
var h = h0; //current height in feet
var v = 0; //current velocity in feet/sec

var oldT = 0; //previous time in seconds
var delT = 0; //change in time in seconds

var oldH = h0; //previous height in feet
var delH = 0; //change in height in feet

//The following variable is used to insert spaces
in
// the output display.
var sp = "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;";

//Compute and display various values for as long
as the
// height of the projectile is above the ground.
while(h > 0){
    //This is the general equation for the height of
a
    // projectile under the influence of gravity in
a
    // vacuum.

$$h = h_0 + v_0 * t + 0.5 * g * t^2;$$


    //Compute and save the time interval since the
    // previous estimate of height, etc.
    delT = t-oldT;
    oldT = t; //Save the current time for the next
iteration

    //Estimate the velocity based on the change in
    // height since the previous estimate.
    delH = h-oldH;
    v = delH/delT; //compute velocity
    oldH = h; //Save the current height for the next
iteration.
```

```

//Display the information for this iteration
document.write(
    "t = " + t.toFixed(2) + " seconds" + sp +
    " h = " + h.toFixed(1) + " feet" + sp +
    " v = " + v.toFixed(2) + " feet/second" + sp
+
    "<br/>");

//Increment the time for the next iteration.
t = t + tInc;
} //end while loop

document.write("End Script");

</script>
</body></html>

```

Screen output

The text shown in [Figure 4](#) should appear in your browser window when the html file opens in your browser.

Figure 4 . Screen output for Listing #2.

Start Script		
t = 0.00 seconds	h = 6.0 feet	v =
100.00 feet/second		
t = 0.25 seconds	h = 30.0 feet	v =
95.97 feet/second		
t = 0.50 seconds	h = 52.0 feet	v =
87.92 feet/second		

Figure 4 . Screen output for Listing #2.

t = 0.75 seconds	h = 72.0 feet	v =
79.87 feet/second		
t = 1.00 seconds	h = 89.9 feet	v =
71.82 feet/second		
t = 1.25 seconds	h = 105.8 feet	v =
63.77 feet/second		
t = 1.50 seconds	h = 119.8 feet	v =
55.72 feet/second		
t = 1.75 seconds	h = 131.7 feet	v =
47.67 feet/second		
t = 2.00 seconds	h = 141.6 feet	v =
39.62 feet/second		
t = 2.25 seconds	h = 149.5 feet	v =
31.57 feet/second		
t = 2.50 seconds	h = 155.4 feet	v =
23.52 feet/second		
t = 2.75 seconds	h = 159.2 feet	v =
15.47 feet/second		
t = 3.00 seconds	h = 161.1 feet	v =
7.42 feet/second		
t = 3.25 seconds	h = 160.9 feet	v =
-0.63 feet/second		
t = 3.50 seconds	h = 158.8 feet	v =
-8.68 feet/second		
t = 3.75 seconds	h = 154.6 feet	v =
-16.73 feet/second		
t = 4.00 seconds	h = 148.4 feet	v =
-24.78 feet/second		
t = 4.25 seconds	h = 140.2 feet	v =
-32.83 feet/second		
t = 4.50 seconds	h = 130.0 feet	v =
-40.88 feet/second		
t = 4.75 seconds	h = 117.7 feet	v =
-48.93 feet/second		
t = 5.00 seconds	h = 103.5 feet	v =

Figure 4 . Screen output for Listing #2.

```
-56.98 feet/second
t = 5.25 seconds      h = 87.2 feet      v =
-65.03 feet/second
t = 5.50 seconds      h = 69.0 feet      v =
-73.08 feet/second
t = 5.75 seconds      h = 48.7 feet      v =
-81.13 feet/second
t = 6.00 seconds      h = 26.4 feet      v =
-89.18 feet/second
t = 6.25 seconds      h = 2.1 feet      v =
-97.23 feet/second
t = 6.50 seconds      h = -24.2 feet     v =
-105.28 feet/second
End Script
```

A graphical representation of velocity

The velocity values shown in [Figure 4](#) are plotted in [Figure 1](#) along with the height values and the acceleration values, which will be computed shortly..

Analysis of the results

The difference between [Figure 2](#) and [Figure 4](#) is the addition of a column of velocity information on the right in [Figure 4](#). As you can see, the velocity begins with a positive value of 100 feet/second at the top of the velocity column.

The velocity goes to zero somewhere between 3 and 3.25 seconds, which is the point where the height of the arrow reaches its maximum.

At 3.25 seconds, the velocity switches from positive to negative meaning that the arrow begins falling back toward the earth. The velocity reaches -100 feet/second somewhere between 6.25 and 6.5 seconds, which is about the same time that the arrow hits the ground.

Hopefully you have recognized that the velocity curve for this particular situation is a straight line with a negative slope. That straight line crosses the horizontal axis going from positive territory into negative territory about half way between the original release of the arrow and the point where the arrow strikes the ground.

Analysis of the code

A comparison of [Listing 2](#) with [Listing 1](#) shows the addition of the following variables:

- `var oldT = 0; //previous time in seconds`
- `var delT = 0; //change in time in seconds`
- `var oldH = h0; //previous height in feet`
- `var delH = 0; //change in height in feet`

Save values for use in next iteration

As you will see when I get into an explanation of the **while** loop, the variables **oldT** and **oldH** are used to save the time value and the height value during each iteration to make those values available during the next iteration.

Compute changes in time and height

Inside the while loop, the current value of **t** and the value stored in **oldT** are used to compute the change in time since the previous iteration. The current value of **h** and the value stored in **oldH** are used to compute the change in height since the previous iteration. These change values are saved in the variables named **delT** and **delH** .

Velocity is the ratio of the changes

In this case, the value in **delH** represents the displacement of the arrow during the time interval and the value stored in **delT** represents the length of the time interval. As you already know, the velocity is the ratio of those two values. The value for velocity is computed along with the time and the height and then the script executes the next iteration of the **while** loop.

The while loop

The **while** loop in [Listing 2](#) begins the same as in [Listing 1](#). After the new height value is computed, the script performs the computations described above to estimate and display the velocity for the time interval since the previous iteration.

In addition, the script saves the current values of **t** and **h** in **oldT** and **oldH** to be used to estimate the velocity during the next iteration of the **while** loop.

Finally, the time value is increased by the value stored in the variable named **tInc**, and control goes back to the top of the loop where the current value of height is tested to determine if it has become negative (meaning that the arrow has hit the ground).

Acceleration of gravity exercise #1

To repeat the earlier definition, acceleration is the rate of change of velocity.

You learned earlier that velocity is the rate of change of position. Then you learned that we can estimate the velocity of a moving object at a point in time by determining the slope of a curve that plots the position of the object versus time at that particular point in time.

Positive and negative velocity

Positive slopes indicate positive velocity and negative slopes indicate negative velocity. In the case of our arrow, the velocity went to zero and switched from positive to negative at the point where the arrow reached its maximum height and began to fall back toward the earth. In a math class, that point would likely be referred to as an **inflection point** on the curve of position versus time.

Estimating acceleration

We can extend that procedure to estimate the acceleration of a moving object at a point in time by determining the slope of a curve that plots the velocity of the object versus time at that particular point in time. Positive slopes indicate positive acceleration and negative slopes indicate negative acceleration.

Estimate some velocity values graphically

If you have the velocity curve from [Figure 4](#) plotted on your graph, I recommend that you estimate and record the slope and hence the acceleration at a few points. That should be easy to do in this case, because the velocity curve should be a straight line with a negative slope.

A constant acceleration value

If the velocity curve is a straight line with a negative slope, and if the acceleration at any point in time is equal to the slope of the velocity curve, that means that the acceleration has a constant negative value throughout the time interval of interest. In fact, you should find that the value of the acceleration is approximately $-32.2 \text{ feet/second}^2$, which is known as the **acceleration of gravity**.

Create a script

Copy the code from [Listing 3](#) into an html file and open that file in your browser.

Listing 3 . Acceleration of gravity exercise #1.

```
<!------- File JavaScript99.html ----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/>");

//Declare and initialize constants
var g = -32.2;//acceleration of gravity in
```

```

ft/(sec*sec)
var v0 = 100;//initial velocity in ft/sec
var h0 = 6;//initial height in feet
var tInc = 0.25;//calculation interval in seconds

//Declare and initialize working variables
var t = .0001;//current time in seconds
var h = h0;//current height in feet
var v = 0;//current velocity in feet/sec
var a = 0;//current acceleration in feet/(sec*sec)

var oldT = 0;//previous time in seconds
var delT = 0;//change in time in seconds

var oldH = h0;//previous height in feet
var delH = 0;//change in height in feet

var oldV = 0;//previous velocity in feet/sec
var delV = 0;//change in velocity in feet/sec

//The following variable is used to insert spaces
in
// the output display.
var sp = "&nbsp;&nbsp;&nbsp;";

//Compute and display various values for as long
as the
// height of the projectile is above the ground.
while(h > 0){
    //This is the general equation for the height of
a
    // projectile under the influence of gravity in
a
    // vacuum.
    h = h0 + v0*t +0.5*g*t*t;

    //Compute and save the time interval since the

```

```

    // previous estimate of height, etc.
    delT = t-oldT;
    oldT = t;//Save the current time for the next
iteration.

    //Estimate the velocity based on the change in
    // height since the previous estimate.
    delH = h-oldH;
    v = delH/delT;//compute velocity
    oldH = h;//Save the current height for the next
iteration.

    //Estimate the acceleration based on the change
in
    // velocity once the data pipeline is full.
    delV = v - oldV;
    oldV = v;//Save the current velocity for the
next iteration.
    if(t > 2*tInc){
        a = delV/delT;//compute acceleration
    }//end if

    //Display the information for this iteration
    document.write(
        "t = " + t.toFixed(2) + " sec" + sp +
        " h = " + h.toFixed(0) + " ft" + sp +
        " v = " + v.toFixed(2) + " ft/sec" + sp +
        " a = " + a.toFixed(2) + " ft/(sec*sec)" +
        "<br/>");

    //Increment the time for the next iteration.
    t = t + tInc;
} //end while loop

document.write("End Script");

```

```
</script>
</body></html>
```

Screen output

The text shown in [Figure 5](#) should appear in your browser window when the html file is opened in your browser.

Figure 5 . Screen output for Listing #3.

```
Start Script
t = 0.00 sec    h = 6 ft    v = 100.00 ft/sec
a = 0.00 ft/(sec*sec)
t = 0.25 sec    h = 30 ft    v = 95.97 ft/sec
a = 0.00 ft/(sec*sec)
t = 0.50 sec    h = 52 ft    v = 87.92 ft/sec
a = -32.20 ft/(sec*sec)
t = 0.75 sec    h = 72 ft    v = 79.87 ft/sec
a = -32.20 ft/(sec*sec)
t = 1.00 sec    h = 90 ft    v = 71.82 ft/sec
a = -32.20 ft/(sec*sec)
t = 1.25 sec    h = 106 ft    v = 63.77 ft/sec
a = -32.20 ft/(sec*sec)
t = 1.50 sec    h = 120 ft    v = 55.72 ft/sec
a = -32.20 ft/(sec*sec)
t = 1.75 sec    h = 132 ft    v = 47.67 ft/sec
a = -32.20 ft/(sec*sec)
t = 2.00 sec    h = 142 ft    v = 39.62 ft/sec
a = -32.20 ft/(sec*sec)
t = 2.25 sec    h = 149 ft    v = 31.57 ft/sec
a = -32.20 ft/(sec*sec)
t = 2.50 sec    h = 155 ft    v = 23.52 ft/sec
```

Figure 5 . Screen output for Listing #3.

```
a = -32.20 ft/(sec*sec)
t = 2.75 sec    h = 159 ft    v = 15.47 ft/sec
a = -32.20 ft/(sec*sec)
t = 3.00 sec    h = 161 ft    v = 7.42 ft/sec
a = -32.20 ft/(sec*sec)
t = 3.25 sec    h = 161 ft    v = -0.63 ft/sec
a = -32.20 ft/(sec*sec)
t = 3.50 sec    h = 159 ft    v = -8.68 ft/sec
a = -32.20 ft/(sec*sec)
t = 3.75 sec    h = 155 ft    v = -16.73 ft/sec
a = -32.20 ft/(sec*sec)
t = 4.00 sec    h = 148 ft    v = -24.78 ft/sec
a = -32.20 ft/(sec*sec)
t = 4.25 sec    h = 140 ft    v = -32.83 ft/sec
a = -32.20 ft/(sec*sec)
t = 4.50 sec    h = 130 ft    v = -40.88 ft/sec
a = -32.20 ft/(sec*sec)
t = 4.75 sec    h = 118 ft    v = -48.93 ft/sec
a = -32.20 ft/(sec*sec)
t = 5.00 sec    h = 103 ft    v = -56.98 ft/sec
a = -32.20 ft/(sec*sec)
t = 5.25 sec    h = 87 ft     v = -65.03 ft/sec
a = -32.20 ft/(sec*sec)
t = 5.50 sec    h = 69 ft     v = -73.08 ft/sec
a = -32.20 ft/(sec*sec)
t = 5.75 sec    h = 49 ft     v = -81.13 ft/sec
a = -32.20 ft/(sec*sec)
t = 6.00 sec    h = 26 ft     v = -89.18 ft/sec
a = -32.20 ft/(sec*sec)
t = 6.25 sec    h = 2 ft      v = -97.23 ft/sec
a = -32.20 ft/(sec*sec)
t = 6.50 sec    h = -24 ft    v = -105.28 ft/sec
a = -32.20 ft/(sec*sec)
End Script
```


A graphical representation of acceleration

The acceleration values shown in [Figure 5](#) are plotted in [Figure 1](#) along with the height values and the velocity values.

Analysis of the results

The main difference between [Figure 5](#) and [Figure 4](#) is the addition of an acceleration column on the right in [Figure 5](#). (In addition, I abbreviated feet and seconds as ft and sec, and reduced the space between columns to make it all fit in the available space.)

Except for the first two values at the top of the acceleration column, every acceleration value is $-32.2 \text{ ft}/(\text{sec} \cdot \text{sec})$. This is what we predicted earlier when we observed that the velocity curve is a straight line with a constant negative slope. (Note that the procedure used to estimate the acceleration did not allow for an accurate estimate of acceleration for the first two values in the acceleration column.)

Free fall

According to [Wikipedia](#), free fall is any motion of a body where gravity is the only force acting upon it.

Since this definition does not specify velocity, it also applies to objects initially moving upward (which is the case with our arrow that was shot upward).

Free fall was demonstrated on the moon by astronaut David Scott on August 2, 1971. He simultaneously released a hammer and a feather from the same height above the moon's surface. The hammer and the feather both fell at the same rate and hit the ground at the same time. This demonstrated Galileo's discovery that in the absence of air resistance, all objects experience the same acceleration due to gravity.

By the way, it is no accident that the value shown for acceleration in [Figure 4](#) matches the value specified for the acceleration of gravity near the beginning of the code in [Listing 3](#).

Analysis of the code

The code in [Listing 3](#) is very similar to the code in [Listing 2](#) with the addition of the code necessary to estimate the acceleration values on the basis of the slope of the velocity curve.

The code that estimates the acceleration in [Listing 3](#) is so similar to the code that estimates the velocity in [Listing 2](#) that I won't bore you by explaining the code in detail.

However, there is one bit of new code that is worthy of note. Without going into detail as to the reasons why, this procedure is incapable of accurately estimating the slope of the velocity curve during the first two time intervals. Therefore, an **if** statement was written into the **while** loop to force the acceleration estimate to be zero for the first two time intervals.

Acceleration of gravity exercise #2

In the previous exercises, the arrow was shot straight up in the air. However, that is rarely the case. Normally, an arrow is shot in an attempt to strike a target some horizontal distance away.

A more realistic scenario

Let's modify our scenario such that the archer shoots the arrow with an initial velocity of 100 feet per second at an angle of 45 degrees relative to the horizontal axis. We will compute and plot the horizontal and vertical position of the arrow at uniform increments in time from the beginning to the end of its trajectory.

Motion of a projectile with uniform acceleration

The following equation describes the straight-line motion of a projectile **with uniform acceleration** .

$$d = v_0 * t + 0.5 * a * t^2$$

where

- d is distance in units of distance
- v_0 is the initial velocity in units of distance/time
- t is time in units of time
- a is acceleration in units of distance/time²

As usual, the units for distance, time, velocity, and acceleration must be consistent with one another.

Shooting the arrow straight up

If we shoot an arrow straight up, as was the case in the previous exercises, the initial velocity doesn't have a horizontal component. Instead, the initial velocity has only a vertical component and the corresponding motion of the arrow has only a vertical component. In other words, the arrow goes straight up and it comes down in the same spot.

Shooting the arrow other than straight up

However, if we shoot the arrow in a direction other than straight up or straight down, the initial velocity has both a vertical component and a horizontal component. Therefore, the resulting motion has both a vertical component and a horizontal component.

To determine the position of the arrow at some time after it is released, we must determine both the vertical and the horizontal component of its motion.

Compute vertical and horizontal components independently

The equation given [above](#) can be used to compute the vertical and horizontal components of motion independently of one another. This approach can be used to give us the position as a function of time along each axis. We can then use that information to determine and plot the position of the arrow in x-y space at one or more points in time.

Vertical and horizontal motion components are independent

It is important to understand that the vertical and horizontal components of motion are independent of one another. In other words, changing the vertical component of motion doesn't effect the horizontal motion and changing the horizontal component of motion doesn't effect the vertical motion. The overall motion seen by an observer is the superposition of the two.

Create a script

Please copy the code shown in [Listing 4](#) into an html file and open the file in your browser.

Listing 4 . Acceleration of gravity exercise #2.

```
<!------- File JavaScript04.html ----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/>");

//Define common parameters
var ang = 45;//firing angle degrees re the
horizontal
var tInc = 0.25;//calculation interval in seconds
var v0 = 100;//initial velocity in ft/sec

//Define horizontal parameters
var ax = 0;//horizontal acceleration
var vx0 = v0*Math.cos(
    ang*Math.PI/180);//component of initial velocity
in ft/sec
var x0 = 0;//initial x-offset in feet

//Define vertical parameters
var ay = -32.2;//vertical acceleration in
ft/sec*sec
var vy0 = v0*Math.sin(
```

```

    ang*Math.PI/180); //component of initial velocity
in ft/sec
var y0 = 6; //initial height in feet

//Declare and initialize working variables
var t = .0001; //current time in seconds
var x = x0; //current x in feet
var y = y0; //current height in feet

//The following variable is used to insert spaces
in
// the output display.
var sp = "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;";

//Compute and display the horizontal and vertical
// positions of the projectile at uniform
increments
// of time
while(y > 0){
    //These are the general equations for the
straight-
    // line motion of a projectile under the
influence
    // of uniform acceleration in a vacuum.
    y = y0 + vy0*t + 0.5*ay*t*t;
    x = x0 + vx0*t + 0.5*ax*t*t;

    //Display the information for this iteration
    document.write(
        "t = " + t.toFixed(2) + " seconds" + sp +
        " x = " + x.toFixed(1) + " feet" + sp +
        " y = " + y.toFixed(1) + " feet" + sp +
        "<br/>");

    //Increment the time for the next iteration.
    t = t + tInc;
} //end while loop

```

```
document.write("End Script");
```

```
</script>
```

```
</body></html>
```

Screen output

The text shown in [Figure 6](#) should appear in your browser when you open the html file in your browser.

Figure 6 . Screen output for Listing #4 at 45 degrees.

Start Script		
t = 0.00 seconds	x = 0.0 feet	y = 6.0
feet		
t = 0.25 seconds	x = 17.7 feet	y =
22.7 feet		
t = 0.50 seconds	x = 35.4 feet	y =
37.3 feet		
t = 0.75 seconds	x = 53.0 feet	y =
50.0 feet		
t = 1.00 seconds	x = 70.7 feet	y =
60.6 feet		
t = 1.25 seconds	x = 88.4 feet	y =
69.2 feet		
t = 1.50 seconds	x = 106.1 feet	y =
75.8 feet		
t = 1.75 seconds	x = 123.8 feet	y =
80.4 feet		
t = 2.00 seconds	x = 141.4 feet	y =
83.0 feet		

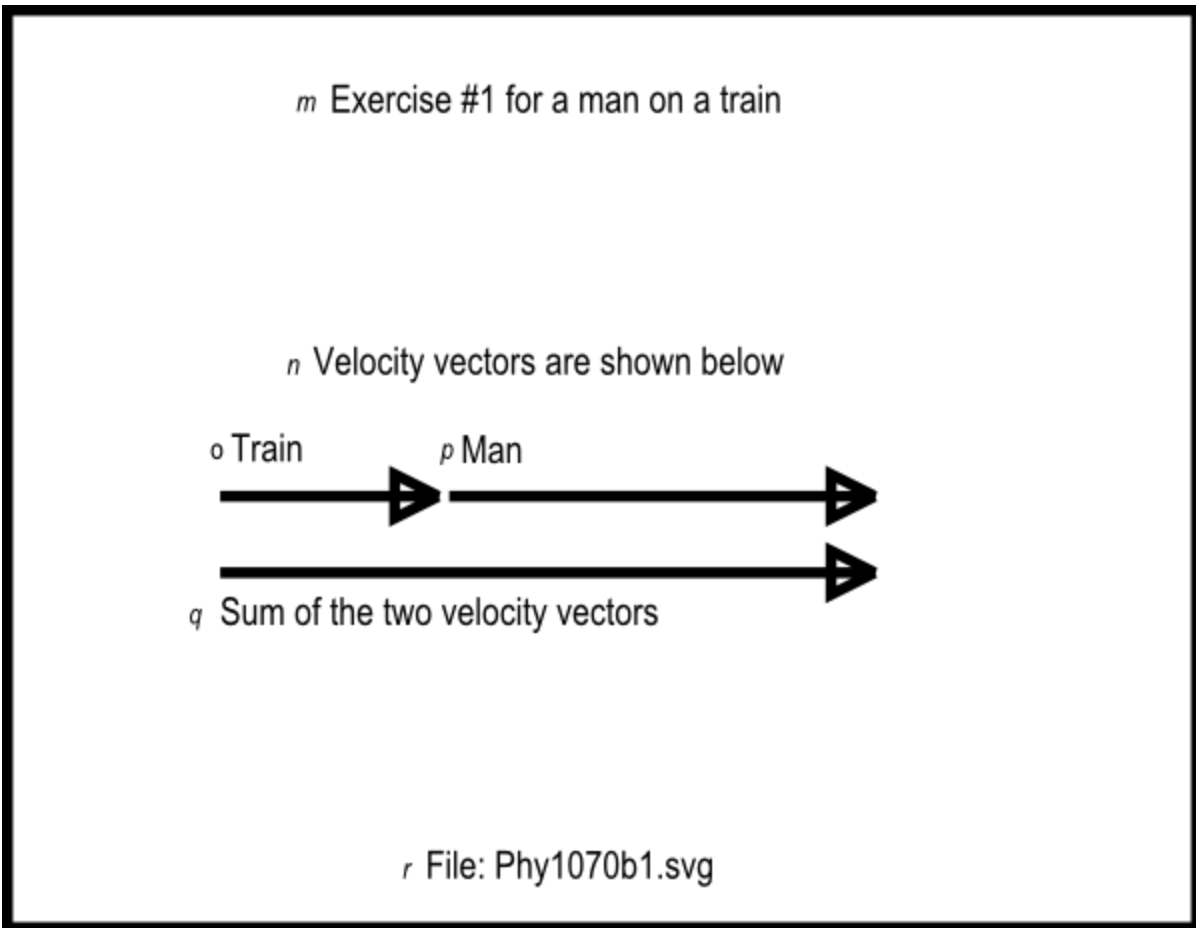
Figure 6 . Screen output for Listing #4 at 45 degrees.

t = 2.25 seconds	x = 159.1 feet	y =
83.6 feet		
t = 2.50 seconds	x = 176.8 feet	y =
82.2 feet		
t = 2.75 seconds	x = 194.5 feet	y =
78.7 feet		
t = 3.00 seconds	x = 212.1 feet	y =
73.2 feet		
t = 3.25 seconds	x = 229.8 feet	y =
65.8 feet		
t = 3.50 seconds	x = 247.5 feet	y =
56.3 feet		
t = 3.75 seconds	x = 265.2 feet	y =
44.8 feet		
t = 4.00 seconds	x = 282.8 feet	y =
31.2 feet		
t = 4.25 seconds	x = 300.5 feet	y =
15.7 feet		
t = 4.50 seconds	x = 318.2 feet	y =
-1.8 feet		
End Script		

A graphical representation of the trajectory

[Figure 7](#) shows the trajectory of the arrow for firing angles of 45 degrees and 60 degrees. The vertical axis shows the height in feet and the horizontal axis shows the flight distance in feet.

Figure 7 - Trajectory for two different firing angles.



Analysis of the output

[Figure 6](#) shows the x and y coordinates of the arrow's position every 0.25 seconds from the time that it is released until it strikes the ground a little less than 4.5 seconds later. This is the data for the case where the firing angle was 45 degrees.

Compute for a different angle

Next, I recommend that you modify the script by changing the value of the variable named **ang** from 45 degrees to 60 degrees and load the modified version of the html file into your browser. This should produce the output shown in [Figure 8](#). This output is also plotted in [Figure 7](#).

Which arrow went higher?

Which arrow went further along the horizontal axis?

Figure 8 . Screen output for Listing #4 at 60 degrees.

```
Start Script
t = 0.00 seconds      x = 0.0 feet      y = 6.0
feet
t = 0.25 seconds      x = 12.5 feet     y =
26.7 feet
t = 0.50 seconds      x = 25.0 feet     y =
45.3 feet
t = 0.75 seconds      x = 37.5 feet     y =
61.9 feet
t = 1.00 seconds      x = 50.0 feet     y =
76.5 feet
t = 1.25 seconds      x = 62.5 feet     y =
89.1 feet
t = 1.50 seconds      x = 75.0 feet     y =
99.7 feet
t = 1.75 seconds      x = 87.5 feet     y =
108.3 feet
t = 2.00 seconds      x = 100.0 feet    y =
114.8 feet
t = 2.25 seconds      x = 112.5 feet    y =
119.4 feet
t = 2.50 seconds      x = 125.0 feet    y =
121.9 feet
t = 2.75 seconds      x = 137.5 feet    y =
122.4 feet
t = 3.00 seconds      x = 150.0 feet    y =
120.9 feet
t = 3.25 seconds      x = 162.5 feet    y =
117.4 feet
t = 3.50 seconds      x = 175.0 feet    y =
111.9 feet
t = 3.75 seconds      x = 187.5 feet    y =
104.3 feet
t = 4.00 seconds      x = 200.0 feet    y =
```

Figure 8 . Screen output for Listing #4 at 60 degrees.

94.8 feet		
t = 4.25 seconds	x = 212.5 feet	y =
83.2 feet		
t = 4.50 seconds	x = 225.0 feet	y =
69.7 feet		
t = 4.75 seconds	x = 237.5 feet	y =
54.1 feet		
t = 5.00 seconds	x = 250.0 feet	y =
36.5 feet		
t = 5.25 seconds	x = 262.5 feet	y =
16.9 feet		
t = 5.50 seconds	x = 275.0 feet	y =
-4.7 feet		
End Script		

Perhaps you could do the same thing for some other launch angles and determine which launch angle provides the greatest horizontal range and which launch angle provides the greatest vertical height.

For example, how far does the arrow fly when launched at an angle of 0 degrees from a height of 6 feet?

Analysis of the code

The code in this script treats the horizontal and vertical components of motion independently and uses the equation given [above](#) to determine the position along that axis as a function of time every 0.25 seconds.

Common parameter values

The code in [Listing 4](#) begins by defining common parameter values for the firing angle, the time increment at which coordinate values will be computed and displayed, and the initial velocity of the arrow in the firing direction.

Horizontal and vertical parameters

Following that, the code in [Listing 4](#) defines horizontal parameters and vertical parameters that will be used to compute the horizontal and vertical components of motion respectively.

Horizontal acceleration

The horizontal acceleration is set to zero. In the absence of air resistance, there is nothing to cause the horizontal component of the arrow to speed up or slow down until it stops suddenly when it hits the ground.

Vertical acceleration

The vertical acceleration is set to the acceleration of gravity at the surface of the earth, which is the same value used for the previous exercises in this module.

Decompose the initial velocity

The cosine and sine functions are used to decompose the initial velocity into horizontal and vertical components of velocity.

Initial horizontal and vertical positions

Finally, the horizontal position when the arrow is released is set to 0 and the vertical position is set to 6 feet, approximately the height of the release point for an archer that is six feet tall.

Working variables

After defining the horizontal and vertical parameters, the code in [Listing 4](#) declares working variables for time, horizontal position (x), and vertical position (y).

A while loop

A **while** loop is used to iterate for as long as the arrow is above the ground.

During each iteration, the equation given [above](#) is evaluated twice, once to determine the height of the arrow and once to determine the horizontal position of the arrow at the current time.

Time starts at 0, and increments by +0.25 seconds during each iteration.

The **document.write** method is called inside the while loop to display the results shown in [Figure 6](#).

Acceleration of gravity exercise #3

A quadratic equation

Now let's turn things around and approach the problem from a different viewpoint. If we subtract **d** from both sides of the motion equation given [above](#), replace **a** by **g** to avoid confusion later, and then rearrange the terms, we get **the following equation** :

$$0.5*g*t^2+v0*t-d = 0$$

where

- d is distance in units of distance
- v0 is the initial velocity in units of distance/time
- t is time in units of time
- g is acceleration in units of distance/time^2

In this form, you may recognize this as a standard quadratic equation which is **often written as**

$$a*t^2 + b*t + c = 0$$

(Now you know why I replaced the acceleration term **a** by the acceleration term **g** earlier. I need to use the symbol **a** in the standard quadratic equation.)

Standard solution for a quadratic equation

Given values for a, b, and c, you should know that the solution for determining the values for t is to find the roots of the equation.

There are two roots (hence two values for t). You should also know that the two roots can be found by evaluating the **quadratic formula** in two forms.

$$t1 = (-b + \text{Math.sqrt}(b*b - 4*a*c)) / (2*a);$$

$$t2 = (-b - \text{Math.sqrt}(b*b - 4*a*c)) / (2*a);$$

Using the solution

Relating the coefficients in the [standard motion equation](#) to the coefficients in [the standard quadratic equation](#) gives us:

- $a = 0.5 * g$
- $b = v_0$
- $c = -d$

The scenario

Let's use the scenario posed in the [first exercise](#) in this module. In this scenario, an archer that is six feet tall shoots an arrow directly upward with a velocity of 100 feet per second. Assume that the arrow is at a height of 6 feet when it leaves the bow. Also ignore the effects of air resistance.

Referring back to Figure 1...

We learned from the table in [Figure 2](#) that the arrow was at a height of 89.9 feet at 1 second on the way up from the surface of the earth, and was at a height of 89.9 feet again on the way down at approximately 5.2 seconds.

Let's pretend that we don't have the table in [Figure 1](#) and write a script that uses the quadratic form of the [standard motion equation](#) along with the [quadratic formula](#) to compute the times that the arrow was at a height of 89.9 feet.

Do it also for the moon

To make the problem even more interesting, let's also cause the script to compute and display the times that the arrow was at a height of 89.9 feet assuming that the archer was standing on the surface of the moon instead of the earth.

An issue involving the initial height

The [standard motion equation](#) can be used to compute the time that the arrow has traveled to any specific height, relative to the height from which it was released. Therefore, since the arrow was released from a height of 6 feet and it initially traveled up, we need to determine the time at which it had traveled 83.9 feet to find the time that it was actually at 89.9 feet.

The JavaScript code

Please copy the code from [Listing 5](#) into an html file and open the file in your browser.

Listing 5 . Acceleration of gravity exercise #3.

```
<!------- File JavaScript05.html ----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/><br/>");

//The purpose of the getRoots function is to
// compute and
// return the roots of a quadratic equation
// expressed in
// the format
//  $a \cdot x^2 + b \cdot x + c = 0$ 
//The roots are returned in the elements of a two-
// element
// array. If the roots are imaginary, the function
// returns NaN for the value of each root.
function getRoots(a,b,c){
```

```

    var roots = new Array(2);
    roots[0] = (-b+Math.sqrt(b*b-4*a*c))/(2*a);
    roots[1] = (-b-Math.sqrt(b*b-4*a*c))/(2*a);
    return roots;
} //end getRoots

//Initialize the problem parameters.
var gE = -32.2; //gravity in ft/sec*sec on Earth
var gM = -32.2*0.167; //gravity in ft/sec*sec on
the Moon
var v0 = 100; //initial velocity in ft/sec
var d0 = 6; //initial height in feet
var d = 89.9-d0; //target height corrected for
initial height

//Equate problem parameters to coefficients in a
// standard quadratic equation.
var a = 0.5*gE;
var b = v0;
var c = -d;

//Solve the quadratic formula for the two times at
which the
// height matches the target height corrected for
the
// initial height.
var roots = getRoots(a,b,c);

//Extract the two time values from the array.
var t1 = roots[0]; //time in seconds
var t2 = roots[1]; //time in seconds

//Display the results for the earth.
document.write("On Earth <br/>");
document.write("Arrow is at " + (d+d0) + " feet at
" +
                t1.toFixed(2) + " seconds" + "

```

```

<br/>");
document.write("Arrow is at " + (d+d0) + " feet at
" +
                t2.toFixed(2) + " seconds" + "<br/>
<br/>");

//Compute and display for the moon
//Adjust the value of a to represent the
acceleration
// of gravity on the moon
a = 0.5*gM;

//Solve the quadratic formula for the two times at
which the
// height matches the target height corrected for
the
// initial height.
roots = getRoots(a,b,c);

//Extract the two time values from the array.
var t1 = roots[0];//time in seconds
var t2 = roots[1];//time in seconds

//Display the results.
document.write("On the Moon <br/>");
document.write("Arrow is at " + (d+d0) + " feet at
" +
                t1.toFixed(2) + " seconds" + "
<br/>");
document.write("Arrow is at " + (d+d0) + " feet at
" +
                t2.toFixed(2) + " seconds" + "<br/>
<br/>");

document.write("End Script");

```



```
</script>
</body></html>
```

Screen output

The text shown in [Figure 9](#) should appear in the browser window when you open the html file in the browser.

Figure 9 . Screen output for Listing #5.

Start Script

On Earth

Arrow is at 89.9 feet at 1.00 seconds

Arrow is at 89.9 feet at 5.21 seconds

On the Moon

Arrow is at 89.9 feet at 0.86 seconds

Arrow is at 89.9 feet at 36.33 seconds

End Script

A function named `getRoots`

The quadratic formula isn't very complicated, but it is fairly tedious and easy to type incorrectly. Therefore, I decided to encapsulate it in a function that we can copy into future scripts saving us the need to type it correctly in the future.

[Listing 5](#) begins with the definition of a function named `getRoots` that receives the parameters `a`, `b`, and `c`, and returns the roots of the quadratic

equation in a two-element array.

Real or imaginary roots

The roots of a quadratic equation can be either real or imaginary. If the roots are imaginary, this function simply returns NaN (not a number) for each root.

The parameters of the problem

Following the definition of the **getRoots** function, [Listing 5](#) declares and initializes several variables to establish the parameters of the problem, such as the acceleration of gravity on the earth and moon, the initial velocity of the arrow, etc.

The computed height versus the target height

The target height for the problem is 89.9 feet. Note that the variable named `d` contains that value less the initial height of 6 feet. Thus, the script will find the time at which the arrow has traveled 83.9 feet on the way up, and the time that it has traveled that same distance on the way back down.

Establish quadratic coefficients

The next three lines of code use the problem parameters to establish values for the standard coefficients of a quadratic equation, `a`, `b`, and `c`, as described [above](#). Note that at this point in the script, the coefficient named **a** is based on the acceleration of gravity on earth. (Later, it will be changed to reflect the acceleration of gravity on the moon.)

Get the roots of the quadratic equation

Then the script calls the **getRoots** function, passing `a`, `b`, and `c` as parameters, and stores the returned array containing the roots in the variable named **roots**.

Following that, the script extracts the roots from the array and displays them as shown by the text in the upper half of [Figure 9](#).

Repeat the process for the moon

Then [Listing 5](#) sets the value of the coefficient named **a** to reflect the acceleration of gravity on the moon, repeats the process, and displays the results in the lower half of [Figure 9](#).

Note that the arrow reaches the target height somewhat quicker on the moon due to the lower acceleration of gravity, and takes much longer to arrive at the same height on the way back down to the surface of the moon. Were we to create a chart similar to [Figure 2](#) for the moon, we would see that the arrow goes much higher before turning around and falling back to the surface of the moon.

Other useful equations

You have learned how to use the following equation to solve various physics problems involving motion in a straight line with uniform acceleration so far in this module.

$$h = v_0 * t + 0.5 * g * t^2$$

where

- h is the distance of the projectile above the surface of the earth in units of distance
- v₀ is the initial velocity of the projectile in units of distance/time
- t is time in seconds
- g is the acceleration of gravity, approximately 9.8 meters per second squared, or approximately 32.2 feet per second squared.

Some physics textbooks also list the following equations **as being important**.

$$v = v_0 + g * t$$

$$v^2 = v_0^2 + 2 * g * h$$

where v is the velocity of the object and the other terms are the same as described [above](#).

Exercise to find the velocity

Let's do an exercise using the first of the two equations given [above](#).

An individual on the surface of the earth shoots an arrow directly upward with a velocity of 100 feet per second. How many seconds elapse before the arrow turns and starts falling towards the surface of the earth. Ignore the effects of air resistance.

Create a script

Please copy the code from [Listing 6](#) into an html file and open the file in your browser.

Listing 6 . Exercise to find the velocity.

Listing 6 . Exercise to find the velocity.

```
<!------- File JavaScript06.html ----
----->
<html><body>
<script language="JavaScript1.3">

document.write("Start Script <br/><br/>");

//Initialize the problem parameters.
var g = -32.2;//gravity in ft/sec*sec on Earth
var v0 = 100;//initial velocity in ft/sec

//Given that  $v = v_0 + g * t$ 
//At what time does the velocity go to zero?

//Rearrange the terms in the equation.
var t = -v0/g;

//Display the results
document.write("Arrow has zero velocity at " +
               t.toFixed(2) + " seconds " + "
<br/>");

document.write("<br/>End Script");

</script>
</body></html>
```

Screen output

The text shown in [Figure 10](#) should appear in your browser window when you open the html file in your browser.

Figure 10 . Screen output for Listing #6.

```
Start Script  
  
Arrow has zero velocity at 3.11 seconds  
  
End Script
```

Analysis of the code

Compared to working through the solutions to the previous exercises, this one seems almost trivial.

After establishing values for the acceleration of gravity and the initial velocity of the arrow, the code in [Listing 6](#) rearranges the first equation given [above](#) and solves for the value of time at which the velocity goes to zero. This is the point in time when the arrow turns from moving up and begins falling back toward the earth.

The results are shown in [Figure 10](#). You should compare this result with [Figure 1](#), which shows that the arrow reaches its maximum height at approximately 3 seconds, which agrees very well with the result shown in [Figure 10](#).

Exercise to find the height

Let's do an exercise using the second of the two equations given [above](#).

An individual that is six feet tall standing on the surface of the earth shoots an arrow directly upward with a velocity of 100 feet per second. What is the maximum height achieved by the arrow before it turns and falls back towards the surface of the earth? Ignore the effects of air resistance.

Create a script

Please copy the code from [Listing 7](#) into an html file and open the file in your browser.

Listing 7 . Exercise to find the height.

```
<!------- File JavaScript07.html -----  
----->  
<html><body>  
<script language="JavaScript1.3">  
  
document.write("Start Script <br/><br/>");  
  
//Initialize the problem parameters.  
var g = -32.2;//gravity in ft/sec*sec on Earth  
var v0 = 100;//initial velocity in ft/sec  
var h0 = 6;//initial height  
  
//Given that  $v^2 = v_0^2 + 2*g*h$   
//What is the maximum height reached by the  
arrow?  
//Note that the maximum height is six feet  
more than  
// the value given by the above equation  
because that  
// equation is based on the point of release.  
  
//The maximum height occurs when the velocity  
goes to zero.  
//Setting the velocity to zero and rearranging  
the terms  
// in the equation gives:
```

Listing 7 . Exercise to find the height.

```
var h = h0 + (-(v0 * v0))/(2*g);

//Display the results
document.write("Arrow reaches maximum height
of " +
               h.toFixed(2) + " feet " + "
<br/>");

document.write("<br/>End Script");

</script>
</body></html>
```

Screen output

The text shown in [Figure 11](#) should appear in your browser window when the html file is opened in your browser.

Figure 11 . Screen output for Listing #7.

```
Start Script

Arrow reaches maximum height of 161.28 feet

End Script
```

Analysis of the code

Once again, compared to working through the previous exercises, this one also seems almost trivial.

After establishing values for the acceleration of gravity, the initial velocity of the arrow, and the height at which the arrow was released, the code in [Listing 7](#) rearranges the second equation given [above](#) and solves for the value of the height (relative to the release point) at which the velocity goes to zero. This is the point in the trajectory where the arrow turns from moving up and begins falling back toward the earth.

Note that in order to get the actual height, it was necessary to add the initial height of 6 feet to the computed height.

Compare the results

The results are shown in [Figure 11](#). You should compare this result with [Figure 2](#), which shows that the arrow reaches its maximum height at approximately 161.1 feet, which agrees very well with the result shown in [Figure 11](#).

Run the scripts

I encourage you to run the scripts that I have presented in this lesson to confirm that you get the same results. Copy the code for each script into a text file with an extension of .html. Then open that file in your browser. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: GAME 2302 Motion -- Variable Velocity and Acceleration
- File: Game0380.htm
- Published: 10/13/12
- Revised: 02/01/16

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module. In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-